Matt's Code

Matt James

Contents

1 Introduction **1**
- 1.1 Setting up the environment . . . 1
  - 1.1.1 Linux . . . 1
  - 1.1.2 Windows . . . 1
  - 1.1.3 MacOS . . . 2
- 1.2 Setting up a virtual environment . . . 2
- 1.3 Some python packages . . . 2

2 Plasma Models **5**
- 2.1 `spicedmodel`: The Scalable Plasma Ion Composition and Electron Density Model . . . 5
  - 2.1.1 Installation . . . 6
  - 2.1.2 Usage . . . 6
- 2.2 `spiced` . . . 7
  - 2.2.1 Installation . . . 7
  - 2.2.2 Usage . . . 8
- 2.3 `HermeanFLRModel`: Model of Mercury's dayside plasma mass density . . . 8
  - 2.3.1 Installation . . . 8
  - 2.3.2 Usage . . . 8
- 2.4 `PyGCPM`: Wrapper for the Global Core Plasma Model . . . 9
  - 2.4.1 Installation . . . 9
  - 2.4.2 Usage . . . 9

3 Field Models **11**
- 3.1 `PyGeopack`: Python wrapper for the Tsyganenko field models . . . 11
  - 3.1.1 Installation . . . 11
  - 3.1.2 Usage . . . 11
- 3.2 `geopack`: C++ wrapper for Tsyganenko field models . . . 12
  - 3.2.1 Installation . . . 13
  - 3.2.2 Usage . . . 13
- 3.3 `libinternalfield`: C++ spherical harmonic model code . . . 13
  - 3.3.1 Installation . . . 13
  - 3.3.2 Usage . . . 14
- 3.4 `vsmodel`: Python based Volland-Stern electric field model for Earth . . . 14
  - 3.4.1 Installation . . . 14
  - 3.4.2 Usage . . . 15
- 3.5 `JupiterMag`: Python wrapper for Jovian field models . . . 15
  - 3.5.1 Requirements . . . 15
  - 3.5.2 Installation . . . 17
  - 3.5.3 Usage . . . 18
- 3.6 `libjupitermag`: C++ library for field tracing in Jupiter's magnetosphere . . . 19
  - 3.6.1 Cloning and Building . . . 19
  - 3.6.2 Usage . . . 20
- 3.7 `con2020`: Python implementation of Jupiter's magnetodisc model . . . 23
- 3.8 con2020 . . . 23
  - 3.8.1 Installation . . . 23
  - 3.8.2 Usage . . . 23
- 3.9 `libcon2020`: C++ implementation of Jupiter's magnetodisc model . . . 25
  - 3.9.1 Building libcon2020 . . . 25
  - 3.9.2 Usage . . . 26

3.9.3 Model Parameters . . . 28
3.10 `jrm33`: The JRM33 model in Python . . . 30
3.10.1 Installation . . . 30
3.10.2 Usage . . . 30
3.11 `jrm09`: The JRM09 model in Python . . . 31
3.12 Installation . . . 31
3.13 Installation . . . 31
3.14 Usage . . . 31
3.15 `vip4model`: The VIP4 model in Python . . . 32
3.16 Installation . . . 32
3.17 Usage . . . 32

4 Spacecraft Data **35**
4.1 `Arase`: Download and read Arase data . . . 35
4.1.1 Installation . . . 35
4.1.2 Downloading Data . . . 35
4.1.3 Position and Tracing . . . 35
4.1.4 Reading Data . . . 36
4.1.5 Current Progress . . . 39
4.2 `RBSP`: Download and read Van Allen Probe data . . . 39
4.3 `cluster`: Download and read Cluster data . . . 40
4.4 `pyCRRES`: Download and read CRRES data . . . 40
4.5 `themissc`: Download and read THEMIS data . . . 40
4.6 `imageeuv`: Download and read IMAGE EUV data . . . 40
4.7 `imagerpi`: Download and read IMAGE RPI data . . . 40
4.8 `imagePP`: Download and read Goldstein's plasmapause dataset . . . 40
4.9 Installation . . . 40
4.10 Usage . . . 40
4.11 `PyMess`: Download and read MESSENGER data . . . 41
4.11.1 Installation . . . 41
4.11.2 Submodules . . . 41
4.12 `FIPSProtonData`: Download and read ANN verified FIPS moments . . . 43
4.12.1 Requirements . . . 43
4.12.2 Installation . . . 43
4.12.3 Usage . . . 43
4.13 `VenusExpress`: Download and read VEX data . . . 45

Chapter 1

Introduction

This document lists a bunch of the GitHub repositories created by me which may be useful to others. Some of these repositories are fairly complete, others are less so. I will do my best to fix and update anything that is buggy or incomplete, please do report bugs in the relevant repositories if you can. If you're feeling particularly helpful - feel free to send pull requests.

Most of the code here is written in Python, some things make use of C++ libraries to do some of the heavy lifting, one is a pretty dodgy Python wrapper of a C++ wrapper of Fortran code... Some of the modules and libraries used here are dependencies of others. In the more complete repos `pip` will take care of dependencies, otherwise some manual installation may be required.

1.1 Setting up the environment

In this section I describe how to set up the environment such that everything *should* pretty much work...

1.1.1 Linux

If running on ALICE/SPECTRE, you will most likely be required to enable the following modules:

```
module load gcc/9.3
module load python/gcc/3.9.10
module load git/2.35.2
```

where exact version numbers may change (use whatever is latest, don't just copy and paste!) and the replacement for SPECTRE/ALICE may have another method for loading these things in for all I know. I also recommend adding those to the end of your `/.bashrc` file so that they load every login, e.g.:

```
echo module load gcc/9.3 >> ~/.bashrc
echo module load python/gcc/3.9.10 >> ~/.bashrc
echo module load git/2.35.2 >> ~/.bashrc
```

The above is unlikely to be necessary on a local Linux installation, instead I would recommend installing `git, gcc, g++, make, gfortran` and `pip3`, e.g. in Ubuntu:

```
sudo apt install git gcc g++ binutils gfortran python3-pip
```

All of the above should allow you to install/run/compile most of my code. I wouldn't recommend using Conda in Linux - I know it has cause some problems/confusion when it comes to linking Python with C/C++ on SPECTRE.

1.1.2 Windows

A fair portion of the code is able to run on Windows - much of the Python code is platform independent and some of the C++ libraries/backends are able to be compiled using Windows. In this case, I *would* actually recommend installing Conda, as it worked for me. The GCC compilers (for C/C++/Fortran) can all be installed easily with TDM-GCC (get the 64-bit version here), just remember to put a tick in the box for "fortran" and "openmp".

1.1.3 MacOS

I managed to install the relevant packages in a virtual Hackintosh once. I don't remember how, perhaps using homebrew. Good luck...

1.2 Setting up a virtual environment

In SPECTRE I never actually bothered with a virtual environment, mistakes were made, headaches may have been avoided had I done so. This step is entirely optional, but somewhat recommended:

```
#create a virtual environment, call it what you want,
#here I call mine "env"
python3 -m venv env
```

Once this has been created, you MUST activate it before running any code, or you will just be running things globally:

```
source env/bin/activate
```

note that I am assuming that `env` exists in the current working directory, if not adjust the path accordingly! If it works, the prompt terminal prmpt should change, e.g:

```
#before:
matt@matt-MS-7B86:~$

source env/bin/activate
#after:
(env) matt@matt-MS-7B86:~$
```

1.3 Some python packages

Here are a list of Python packages which are either going to be required by most of my code, or would just be recommended:

1. ipython : best Python interpreter, forget notebooks
2. numpy : essential, don't skip
3. matplotlib : for plotting
4. scipy : loads of good stuff here
5. wheel : used to build Python packages to be installed by pip
6. cdflib : reads CDF files
7. keras : nice for machine learning
8. tensorflow : also machine learning

install them:

```
#update pip first
python3 -m install pip --upgrade --user

pip3 install ipython numpy matplotlib scipy wheel cdflib keras tensorflow --user
```

where the "'`--user`"' flag may or may not be necessary, depending on your version of Python - it places the installed modules in ∼`/.local/lib/python3.9/site-packages`.

In theory, at this point you should be able to run `ipython3` (or just `ipython`) within the terminal, from which any installed code can be imported. The reason I recommend using Ipython over the standard Python interpreter is that it has autocomplete and it uses pretty colours for syntax highlighting. It would also be a good idea to enable the autoreload feature in Ipython, which recompiles anything that has been edited since it was last run, otherwise would have to reload the code manually (or restart the session) after every edit. Run

```
ipython profile create
```

then add the following lines to ~/.ipython/profile_default/ipython_config.py:

```
c.InteractiveShellApp.extensions = ['autoreload']
c.InteractiveShellApp.exec_lines = ['%autoreload 2']
c.InteractiveShellApp.exec_lines.append('print("Warning: disable autoreload in ipython_config.py to imp
```

That should just about do it.

Chapter 2

Plasma Models

2.1 `spicedmodel`: The Scalable Plasma Ion Composition and Electron Density Model

Python wrapper for the Scalable Plasma Ion Composition and Electron Density (SPICED) model:

James, M. K., Yeoman, T.K., Jones, P., Sandhu, J. K., Goldstein, J. (2021), The Scalable Plasma Ion Composition and Electron Density (SPICED) model for Earth's inner magnetosphere, *J. Geophys. Res. Space Physics*, https://doi.org/10.1029/2021JA029565

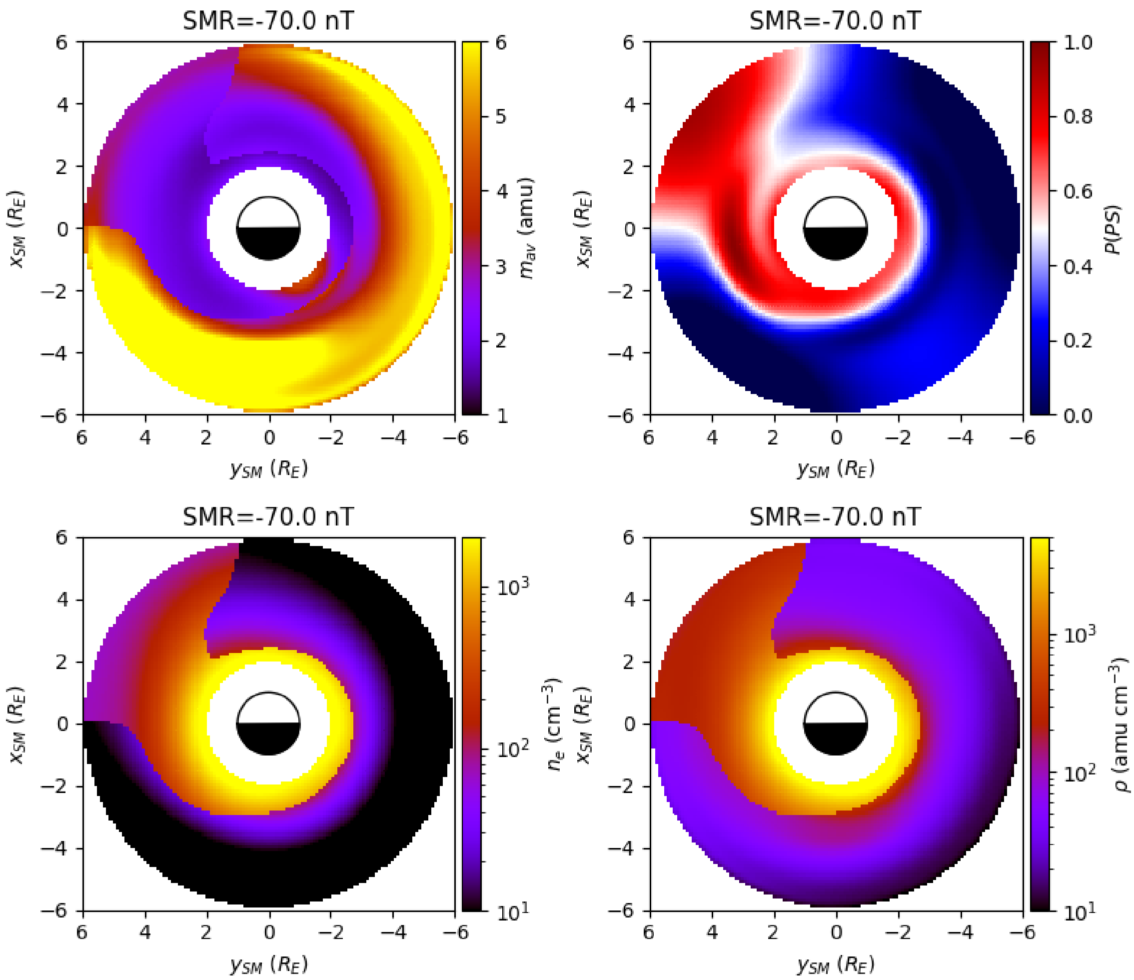


Figure 2.1: Example output of the SPICED model.

2.1.1 Installation

Using pip

This will download the package from PyPI:

```
pip3 install spicedmodel --user
```

From Source

Obtain the latest release from https://github.com/mattkjames7/spicedmodel

```
git clone https://github.com/mattkjames7/spicedmodel
cd spicedmodel
```

Either install using `setup.py`:

```
python3 setup.py install --user
```

or by building a wheel:

```
python3 setup.py bdist_wheel
pip3 install dist/spicedmodel-XXX.whl --user
```

where `"XXX"` is the rest of the file name, which will vary depending upon the current version.

2.1.2 Usage

Load `python3` or `ipython3`, and import

```
import spicedmodel
```

Accessing the models

There are four models, plus two additional combinations of these models:

- Plasmasphere average ion mass, $m_{av,ps}$: `spicedmodel.MavPS`
- Plasmatrough average ion mass, $m_{av,pt}$: `spicedmodel.MavPT`
- Combined average ion mass, m_{av}: `spicedmodel.Mav`
- Hot average ion mass, m_{av}: `spicedmodel.MavHot`
- Probability of being within the plasmasphere, P: `spicedmodel.Prob`
- Plasmasphere electron density, $n_{e,ps}$: `spicedmodel.PS`
- Plasmatrough electron density, $n_{e,pt}$: `spicedmodel.PT`
- Combined electron density, n_e (a combination of plasmasphere, plasmatrough and probability models): `spicedmodel.Density`
- Combined plasma mass density, ρ: `spicedmodel.PMD`

The average versions of each model can be accessed simply by providing the positions in the equatorial plane where you would like them, e.g.:

```
#either using SM x and y coordinates
P = spicedmodel.Prob(x,y)

#or using MLT (M) and L-Shell (L)
P = spicedmodel.Prob(M,L,Coord='ml')
```

The scaled models can be accessed using the same functions, this time including the `SMR` keyword (for `Mav`, `MavPS`, `MavPT`, `Prob`, `PS`, `PT`, `Density` or `PMD`) or `F107` (for `MavHot`), e.g.:

```
#electron density
ne = spicedmodel.Density(x,y,SMR=-75.0)

#average ion mass
mav = spicedmodel.Mav(x,y,SMR=-75.0)

#plasma mass density, effectively ne*mav
pmd = spicedmodel.PMD(x,y,SMR=-75.0)
```

Plotting the models

A simple function is included, `PlotEq`, which allows the plotting of any of the models in the equatorial plane, e.g.:

```
ax = spicedmodel.PlotEq(ptype,SMR=-75.0)
```

where `ptype` is used to tell the function which model to plot, available options are: `'mav'|'mavps'|'mavpt'|'mavhot'|'p`
The following code produces a plot with all 6 models when SMR = -75 nT

```
import matplotlib.pyplot as plt
import spicedmodel

#create the plot window
plt.figure(figsize=(8,7))

#set the parameters of the models
smr = -70.0

#plot the average ion mass
ax0 = spicedmodel.PlotEq('mav',SMR=smr,fig=plt,maps=[2,2,0,0])

#plot probability
ax1 = spicedmodel.PlotEq('prob',SMR=smr,fig=plt,maps=[2,2,1,0])

#plot electron density
ax4 = spicedmodel.PlotEq('density',SMR=smr,fig=plt,maps=[2,2,0,1])

#plot plasma mass density
ax5 = spicedmodel.PlotEq('pmd',SMR=smr,fig=plt,maps=[2,2,1,1])

#adjust everything to fit
plt.tight_layout()
```

2.2 spiced

GitHub: https://github.com/mattkjames7/spiced.git
The C++ code behind the SPICED model. It should be possible to build this library in Linux, Windows and MacOS.

2.2.1 Installation

In Linux and MacOS, it should eb possible to make and install the library:

```
make

#optionally install globally
sudo make install
```

Or in Windows:

```
compile.bat
```

2.2.2 Usage

When using this library, the header file should be included, i.e.:

```
#include <spiced.h>
```

and the linker flag `-lspiced` should be used during compilation.

The models also need to be initialized at runtime using `initModels();`. `spiced.h` contains a full list of the functions which can be linked to using C and other languages like Python within the `extern "C"` {} section; other symbols outside this section can, such as the model objects themselves may be interacted with directly using C++.

2.3 `HermeanFLRModel`: Model of Mercury's dayside plasma mass density

GitHub: https://github.com/mattkjames7/HermeanFLRModel.git

Estimate the dayside plasma mass density in Mercury's magnetosphere using the field line resonance (FLR) based model from James2019.

2.3.1 Installation

This hasn't been placed on PyPI, so either download fromt he GitHub page and install using `pip`, or clone and build the package:

```
#if you download the package
pip3 install HermeanFLRModel-x.y.z-py3-none-any.whl --user

#or clone, build and install
git clone https://github.com/mattkjames7/HermeanFLRModel.git
cd HermeanFLRModel
python3 setup.py bdist_wheel
pip3 install dist/HermeanFLRModel-x.y.z-py3-none-any.whl --user
```

where `x.y.z` should be replaced with the current version number.

2.3.2 Usage

Import the module and create an instance of the `Model` object:

```
import HermeanFLRModel as hflr

model = hflr.Model(Alpha,Coord='MSM')
```

where `Alpha` is the power law index which should be an integer from 0 to 6, and `Coord` sets the coordinate system to use (either MSM or MSO).

Use the `model.Calc()` member function to work out densities:

```
#create input coordinate(s)
x = np.zeros(6)
y = np.array([1.0,1.2,1.4,1.6,1.8,2.0])
z = np.zeros(6)

#call model Calc() function
rho = model.Calc(x,y,z)
```

Or produce a plot of the plasma mass denity for a slice through the magnetosphere, e.g.:

```
#select magnetic local time and alpha
MLT = 6.0
Alpha = 3.0

#plot it
ax = hflr.PlotModelSlice(MLT,Alpha)
```

which should produce something like figure 2.2.

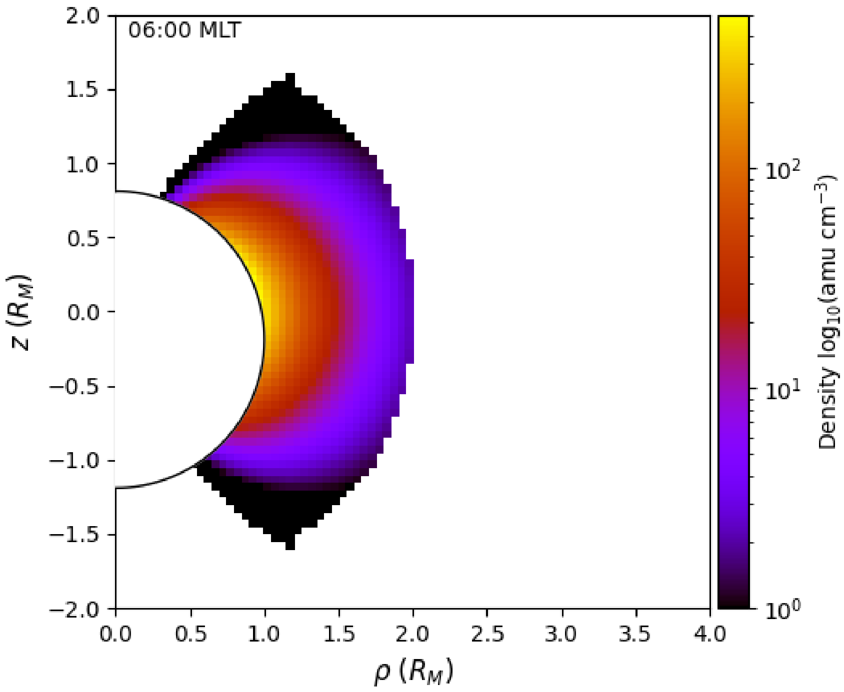


Figure 2.2: Plasma mass density at 6:00 MLT.

2.4 PyGCPM: Wrapper for the Global Core Plasma Model

GitHub: https://github.com/mattkjames7/PyGCPM.git

This is a Python wrapper for the Global Core Plasma Model (Gallagher2000, code found here)

2.4.1 Installation

This module exists on PyPI, so can be installed using `pip`:

```
pip3 install PyGCPM --user
```

2.4.2 Usage

There are three functions:

1. `PyGCPM.GCPM()`: provides particle densities at positions defined in SM coordinates.
2. `PyGCPM.PlotEqSlice()` : plots the density of a species in the equatorial plane.
3. `PyGCPM.PlotMLTSlice()` : plots the density of a particle species in

Firstly, get some densities at some positions in SM coordinates, units of R_E:

```
import PyGCPM
ne,nH,nHe,nO = PyGCPM.GCPM(x,y,z,Date,ut,Kp=Kp,Verbose=Verbose)
```

where `Date` is the date in the format yyyymmdd, `ut` is the time in hours, `Kp` is the Kp index and `Verbose=True` would display progress. The outputs of this function `ne`, `nH`, `nHe` and `nO` are the densities of electrons, protons, helium ions and oxygen ions, respectively.

We can plot the density of a particle species in the equatorial plane:

```
PyGCPM.PlotEqSlice(20010902,12.0,Parameter='ne')
```

which should produce the plot in figure 2.3.

We can also plot the density of a particle species in a slice of MLT:

```
PyGCPM.PlotMLTSlice(8.0,20010902,12.0,Parameter='ne')
```

which should produce the plot in figure 2.4.

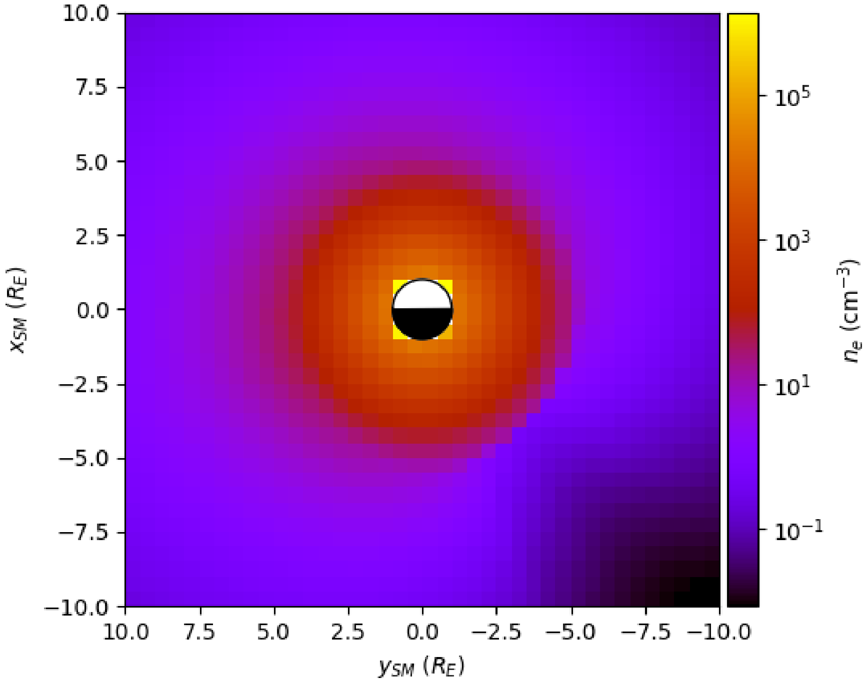


Figure 2.3: Equatorial electon density.

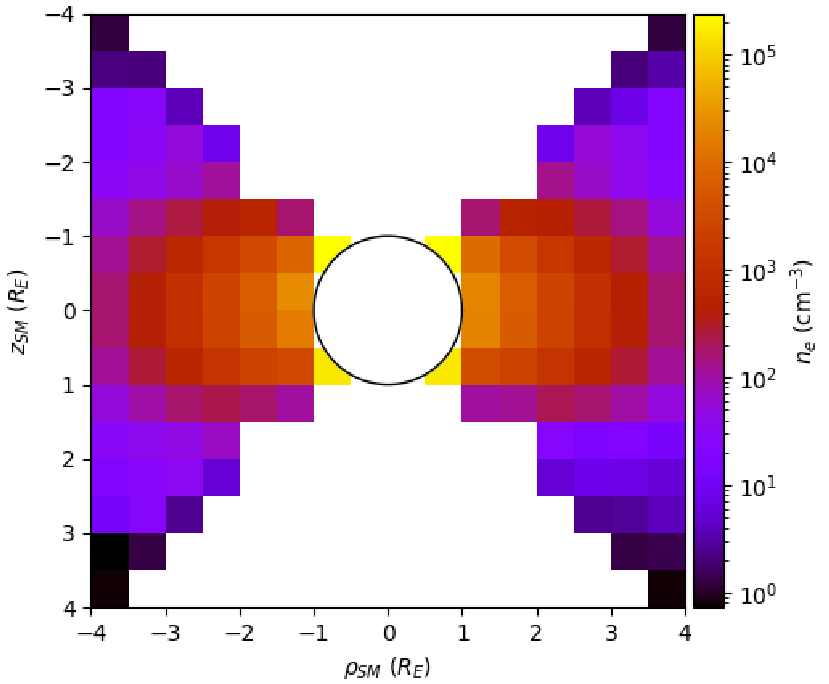


Figure 2.4: MLT slice of electon density.

Chapter 3

Field Models

3.1 PyGeopack: Python wrapper for the Tsyganenko field models

This is a Python module for obtaining field vectors and traces of the Tsygenenko field models. It is a wrapper of a wrapper (see 3.2). The latest code can be viewed and downloaded from here: https://github.com/mattkjames7/PyGeopack.

3.1.1 Installation

The easiest way to install `PyGeopack` is using `pip`, e.g.:

```
pip3 install PyGeopack --user
```

for other installation methods, see the GitHub repo.

At this point, it *may* just work if you were to try to import it, but there's a reasonably good chance that the C++/Fortran code will need to be recompiled, in which case we need to ensure that there are compilers available to do this (see section sectSetup).

One of the features of PyGeopack is that it can easily package together all of the geomagnetic/solar wind parameters that the models use so that when you request a trace or a field vector for a specific date and time, it will automatically try to find the appropriate parameters. This isn't strictly necessary for the models to work, as they will default to some fairly average parameters and they can be overridden manually. In order to be able to use this functionality, this module and the submodules which it relies on to collect the relevant data need to know where they can store the parameters. This means exporting a few environment variables (e.g. in ∼`/.bashrc`):

```
export KPDATA_PATH=/path/to/kp
export OMNIDATA_PATH=/path/to/omni
export GEOPACK_PATH=/path/to/geopack/data
```

which are set as follows for me on SPECTRE:

```
export KPDATA_PATH="/data/sol-ionosphere/mkj13/Kp"
export OMNIDATA_PATH="/data/sol-ionosphere/mkj13/OMNI"
export GEOPACK_PATH="/data/sol-ionosphere/mkj13/Geopack"
```

Once that is done, it should work...

3.1.2 Usage

The first time this is imported, there is a good chance that it will attempt to recompile itself. There will be a lot of messages on the screen, but it should finish successfully. If it fails, double check that you have the required compilers installed, raise an issue on the GitHub page if the problem persists.

If you would lke the latest model parameters, run the following:

```
import PyGeopack as gp
gp.Params.UpdateParameters(SkipWParameters=True)
```

This may take a little time, depending on how much data it needs to download. It will take all of the data and compile it into one binary file ∼350 MiB in size, once this is done, it should be relatively quick loading the data into memory.

The model field vectors can be returned using the `ModelField` function:

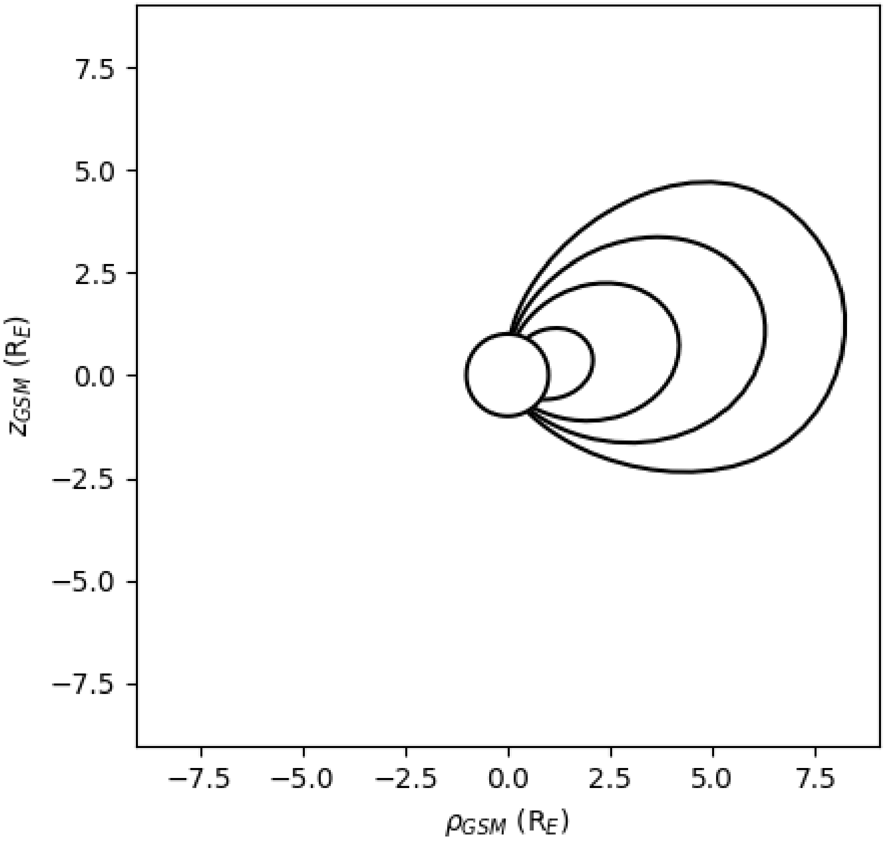


Figure 3.1: Example of field tracing using `PyGeopack`.

```
Bx,By,Bz = gp.ModelField(x,y,z,Date,ut,Model='T96',CoordIn='GSM',**kwargs)
```

where `x`, `y` and `z` can be scalars or arrays of position, in units of Earth radii and in the coordinate system defined by the `CoordIn` keyword (`'GSE'|'GSM'|'SM'`). `Date` can be an array or scalar of date(s) in the format yyyymmdd, while `ut` is in hours from the start of the day. The models currently available are `'T89'`, `'T96'`, `'T01'` and `'TS05'`.

Traces are simple to produce and can be done a single trace at a time, or in batches, e.g.:

```
import numpy as np

#define a few starting positions for the traces
x = np.array([2.0,4.0,6.0,8.0])
y = np.array([0.0,0.0,0.0,0.0])
z = np.array([0.0,0.0,0.0,0.0])

#run the traces, return TraceField object
T = gp.TraceField(x,y,z,20221222,16.0)

#plot field traces
ax = T.PlotRhoZ()
```

which should produce a plot similar to figure 3.1.

For more information on the keyword arguments please see the readme.

3.2 geopack: C++ wrapper for Tsyganenko field models

This is the code that `PyGeopack` calls, it provides a simple C-compatible interface for calculating field vectors, tracing and coordinate conversion. The C++ code in this library is used for configuring model parameters, determining footprints and providing a simple interface for the models, while the Fortran code currently provides field vectors, coordinate transforms and tracing.

3.2.1 Installation

This code can be installed as a linkable library (at least on POSIX systems):

```
#clone the repo
git clone https://github.com/mattkjames7/geopack --recurse-submodules

#cd into it
cd geopack

#fetch the submodules if you forgot the
#--recurse-submodules flag on the clone command
git submodule update --init --recursive

#compile it
make
sudo make install
```

On Linux and MacOS the `sudo make install` command will place the header file at `/usr/local/include/geopack.h` and the shared object file at `/usr/local/lib/libgeopack.so`, unless the `PREFIX` keyword is set (by default `PREFIX=/usr/local`). If the `PREFIX` uses a custom path, then be sure to let the linker know where it exists, either by setting `LD_LIBRARY_PATH` or by using `-L` and `-I`.

On Windows, run `compile.bat` to create `lib`
`libgeopack.dll`.

3.2.2 Usage

To use this code with C and C++, the header file must be included, i.e.:

```
#include <geopack.h>
```

and when compiling, the `-lgeopack` flag should be used to link to the library.

C and other languages such as Python should use the wrapper functions defined within the `extern "C"` {} section of `geopack.h`. This includes `ModelField()` for calculating model field vectors, `TraceField()` for field traces and a number of functions for coordinate conversion, e.g. `SMtoGSMUT()`. An example of how to trace a field line is in `geopack.c`.

C++ is able to use all of the functions declared in `geopack.h`, including the wrapper functions used by C. This means that direct usage of the `Trace` class is possible, an example of this is shown in `geopack.cc`.

3.3 `libinternalfield`: C++ spherical harmonic model code

This library provides field vectors for spherical harmonic magnetic field models. It has a bunch of built in models from magnetized planets and a moon (Ganymede). This library is used by libjupitermag.

3.3.1 Installation

Compile and install in Linux and MacOS:

```
#clone the repo
git clone https://github.com/mattkjames7/libinternalfield

#cd into it
cd libjupitermag

#compile it
make
sudo make install
```

Or in Windows, run `compile.bat` to create `libinternalfield.dll`.

3.3.2 Usage

To use this library, the header should be included `include <internalfield.h>` and the code should be compiled with the `-linternalfield` flag in order to link to the library.

In C, we can use the `getModelFieldPointer()` to get a function pointer to the model we want to use, e.g.:

```
#include <stdio.h>
#include <internalfield.h>

int main() {

        printf("Testing C\n");

        /* try getting a model function */
        modelFieldPtr model = getModelFieldPtr("jrm33");
        double x = 10.0;
        double y = 10.0;
        double z = 0.0;
        double Bx, By, Bz;
        model(x,y,z,&Bx,&By,&Bz);

        printf("B = [%6.1f,%6.1f,%6.1f] nT at [%4.1f,%4.1f,%4.1f]\n",Bx,By,Bz,x,y,z);

        printf("C test done\n");

}
```

C++ can use the `internalModel` object, e.g.:

```
#include <internal.h>

int main() {
    /* set current model */
    internalModel.SetModel("jrm09");

    /* set intput and output coordinates to Cartesian */
    internalModel.SetCartIn(true);
    internalModel.SetCartOut(true);

    /* input position (cartesian)*/
    double x = 35.0;
    double y = 10.0;
    double z = -4.0;

    /* output field */
    double Bx, By, Bz;
    internalModel.Field(x,y,z,&Bx,&By,&Bz);
}
```

3.4 vsmodel: Python based Volland-Stern electric field model for Earth

This is a purely Python-based implementation of the Volland-Stern electric field model Volland1973,Stern1975.

3.4.1 Installation

Use `pip`:

```
pip3 install vsmodel --user
```

3.4.2 Usage

Electric field vectors can be determined using either cylindrical (`vsmodel.ModelE`) or Cartesian (`vsmodel.ModelCart`) coordinates (Solar Magnetic), e.g.:

```
import vsmodel

##### The simple model using Maynard and Chen ####
#the Cartesian model
Ex,Ey,Ez = vsmodel.ModelCart(x,y,Kp)

#the cylindrical model
Er,Ep,Ez = vsmodel.ModelE(r,phi,Kp)


#### The Goldstein et al 2005 version ####
#the Cartesian model, either by providing solar wind
#speed (Vsw) and IMF Bz (Bz), or the equivalent E field (Esw)
Ex,Ey,Ez = vsmodel.ModelCart(x,y,Kp,Vsw=Vsw,Bz=Bz)
Ex,Ey,Ez = vsmodel.ModelCart(x,y,Kp,Esw=Esw)

#the cylindrical model
Er,Ep,Ez = vsmodel.ModelE(r,phi,Kp,Vsw=Vsw,Bz=Bz)
Er,Ep,Ez = vsmodel.ModelE(r,phi,Kp,Esw=Esw)
```

where the Maynard1975 method uses the Kp index and the Goldstein2005 method uses Kp, solar wind speed and the z-component of the interplanetary magnetic field.

The electric field magnitude and $\mathbf{E} \times \mathbf{B}$ velocity can be plotted in the Earth's equatorial plane:

```
import vsmodel
import matplotlib.pyplot as plt

plt.figure(figsize=(9,8))
ax0 = vsmodel.PlotModelEq('E',Kp=1.0,Vsw=-400.0,Bz=-2.5,
                maps=[2,2,0,0],fig=plt,fmt='%4.2f',scale=[0.01,10.0])
ax1 = vsmodel.PlotModelEq('E',Kp=5.0,Vsw=-400.0,Bz=-2.5,
                maps=[2,2,1,0],fig=plt,fmt='%4.2f',scale=[0.01,10.0])
ax2 = vsmodel.PlotModelEq('V',Kp=1.0,Vsw=-400.0,Bz=-2.5,
                maps=[2,2,0,1],fig=plt,scale=[100.0,10000.0])
ax3 = vsmodel.PlotModelEq('V',Kp=5.0,Vsw=-400.0,Bz=-2.5,
                maps=[2,2,1,1],fig=plt,scale=[100.0,10000.0])
ax0.set_title('$K_p=1$; $E_{sw}=-1$ mV m$^{-1}$')
ax2.set_title('$K_p=1$; $E_{sw}=-1$ mV m$^{-1}$')
ax3.set_title('$K_p=5$; $E_{sw}=-1$ mV m$^{-1}$')
ax1.set_title('$K_p=5$; $E_{sw}=-1$ mV m$^{-1}$')
plt.tight_layout()
```

which would produce figure 3.2

3.5 JupiterMag: Python wrapper for Jovian field models

GitHub: https://github.com/mattkjames7/JupiterMag.git

Python wrapper for a collection of Jovian magnetic field models written in C++ (see libjupitermag).

This is part of a community code project : Magnetospheres of the Outer Planets Group Community Code

3.5.1 Requirements

For the Python code to run (without rebuilding the C++ backend), the following Python packages would be required:

- NumPy
- Matplotlib

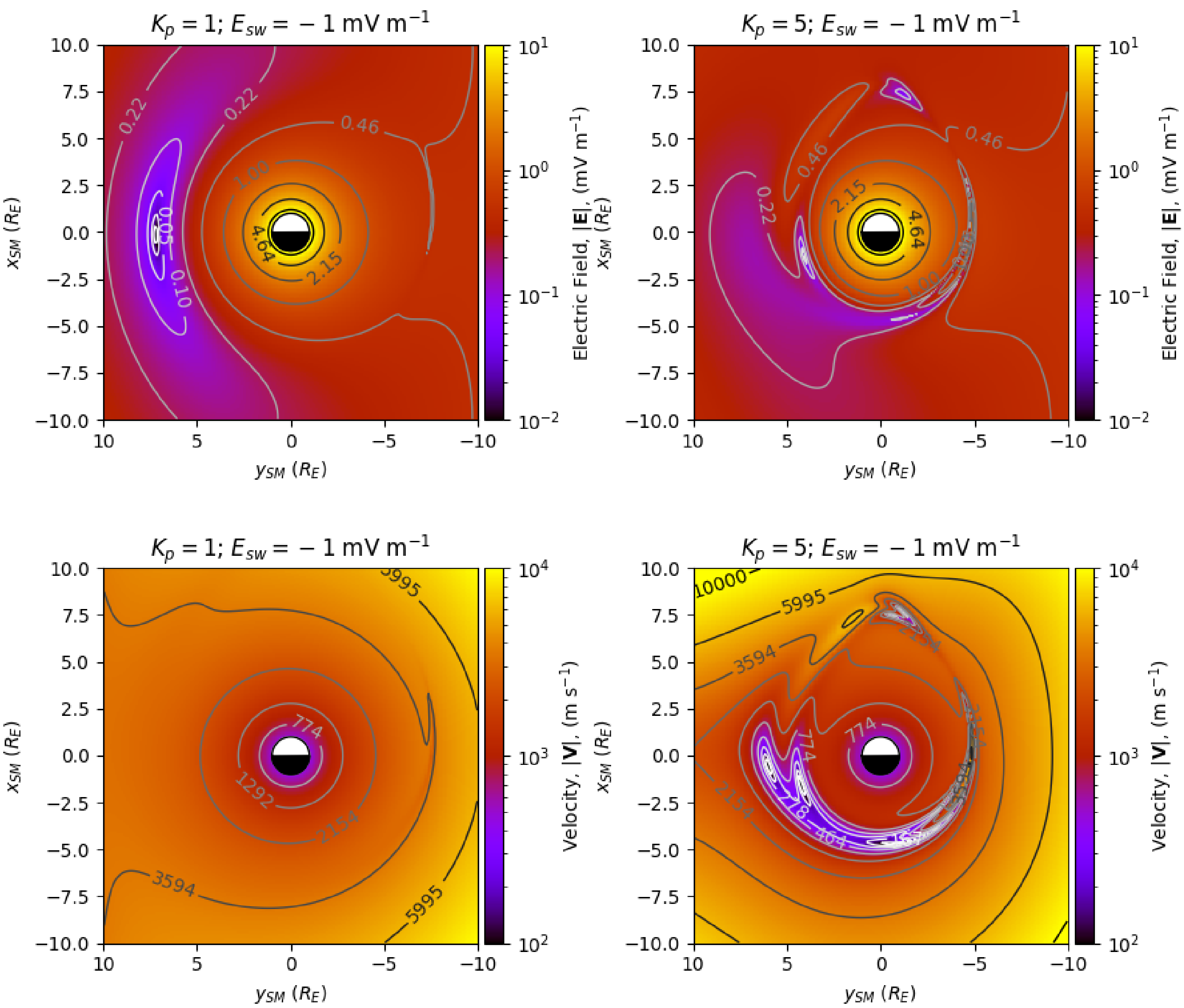


Figure 3.2: Volland-Stern electric field (top plots) and $\mathbf{E} \times \mathbf{B}$ velocity (bottom plots).

- DateTimeTools
- RecarrayTools
- PyFileIO

All of which would be installed automatically if using `pip`.

On some systems, the shared object files would need rebuilding before they can be loaded and accessed using Python. Upon the first import of the `JupiterMag` module, if the shared object/DLL fails to load then it will attempt to use a local C++ compiler to rebuild the binaries.

Linux

JupiterMag was built and tested primarily using Linux Mint 20.3 (based on Ubuntu 20.04/Debian). To rebuild the code, ensure that `g++`, `make`, and `ld` are installed.

Windows

This has been tested on Windows 10 (64-bit), other versions may also work. Requires `g++` and `ld` to work (these can be provided by TDM-GCC). This may or may not work with other compilers installed.

MacOS

This module has been tested on MacOS 11 Big Sur. It requires `g++`, `make`, and `libtool` to recompile (provided by Xcode).

3.5.2 Installation

Install using `pip3`:

```
pip3 install JupiterMag --user
```

Download the latest release (on the right → if you're viewing this on GitHub), then from within the directory where it was saved:

```
pip3 install JupiterMag-1.0.0-py3-none-any.whl --user
```

Or using this repo (replace "1.0.0" with the current version number):

```
#pull this repo
git clone https://github.com/mattkjames7/JupiterMag.git
cd JupiterMag

#update the submodule
git submodule update --init --recursive

#build the wheel file
python3 setup.py bdist_wheel
#the output of the previous command should give some indication of
#the current version number. If it's not obvious then do
# $ls dist/ to see what the latest version is
pip3 install dist/JupiterMag-1.0.0-py3-none-any.whl --user
```

I recommend installing `gcc` $\geq$ 9.3 (that's what this is tested with, earlier versions may not support the required features of C++).

This module should now work with both Windows and MacOS.

Update an Existing Installation

To update an existing installation:

```
pip3 install JupiterMag --upgrade --user
```

Alternatively, uninstall then reinstall, e.g.:

```
pip3 uninstall JupiterMag
pip3 install JupiterMag --user
```

3.5.3 Usage

Internal Field

A number of internal field models are included (see here for more information) and can be accessed via the `JupiterMag.Internal` submodule, e.g.:

```
import JupiterMag as jm

#configure model to use VIP4 in polar coords (r,t,p)
jm.Internal.Config(Model="vip4",CartesianIn=False,CartesianOut=False)
Br,Bt,Bp = jm.Internal.Field(r,t,p)

#or use jrm33 in cartesian coordinates (x,y,z)
jm.Internal.Config(Model="jrm33",CartesianIn=True,CartesianOut=True)
Bx,By,Bz = jm.Internal.Field(x,y,z)
```

All coordinates are either in planetary radii (x, y, z, r) or radians (t, p). All Jovian models here use $R_j = 71,492$ km.

External Field

Currently, the only external field source included is the Con2020 field (see here for the standalone Python code and here for more information on the C++ code used here as part of libjupitermag), other models could be added in the future.

This works in a similar way to the internal field, e.g.:

```
#configure model
jm.Con2020.Config(equation_type='analytic')
Bx,By,Bz = jm.Con2020.Field(x,y,z)
```

Tracing

Field line tracing can be done using the `TraceField` object, e.g.

```
import JupiterMag as jm

#configure external field model prior to tracing
#in this case using the analytic Con2020 model for speed
jm.Con2020.Config(equation_type='analytic')

#trace the field in both directions from a starting position
T = jm.TraceField(5.0,0.0,0.0,IntModel='jrm09',ExtModel='Con2020')
```

The above example will trace the field line from the Cartesian SIII position (5.0,0.0,0.0) (R_j) in both directions until it reaches the planet using the JRM09 internal field model with the Con2020 external field model. The object returned, `T`, is an instance of the `TraceField` class which contains the positions and magnetic field vectors at each step along the trace, along with some footprint coordinates and member functions which can be used for plotting.

A longer example below can be used to compare field traces using just an internal field model (JRM33) with both internal and external field models (JRM33 + Con2020):

```
import JupiterMag as jm
import numpy as np

#be sure to configure external field model prior to tracing
jm.Con2020.Config(equation_type='analytic')
#this may also become necessary with internal models in the future, e.g.
#setting the model degree

#create some starting positions
n = 8
theta = (180.0 - np.linspace(22.5,35,n))*np.pi/180.0
r = np.ones(n)
```

```
x0 = r*np.sin(theta)
y0 = np.zeros(n)
z0 = r*np.cos(theta)

#create trace objects, pass starting position(s) x0,y0,z0
T0 = jm.TraceField(x0,y0,z0,Verbose=True,IntModel='jrm33',ExtModel='none')
T1 = jm.TraceField(x0,y0,z0,Verbose=True,IntModel='jrm33',ExtModel='Con2020')

#plot a trace
ax = T0.PlotRhoZ(label='JRM33',color='black')
ax = T1.PlotRhoZ(fig=ax,label='JRM33 + Con2020',color='red')

ax.set_xlim(-2.0,15.0)
ax.set_ylim(-6.0,6.0)
```

The resulting objects T0 and T1 store arrays of trace positions and magnetic field vectors along with a bunch of footprints. The above code produces figure 3.3.

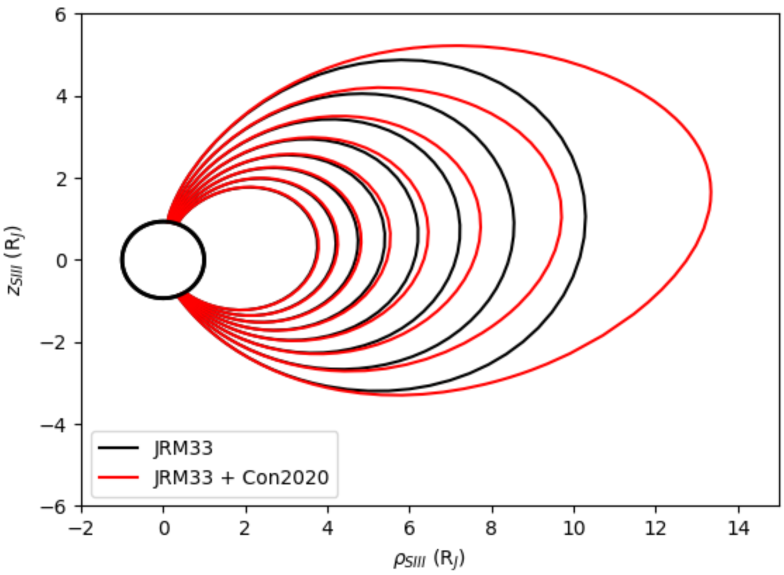


Figure 3.3: Comparison between traces using only the internal field (black) and traces also using a magnetodisc model (red).

3.6 `libjupitermag`: C++ library for field tracing in Jupiter's magnetosphere

GitHub: https://github.com/mattkjames7/libjupitermag.git

Code for obtaining magnetic field vectors and traces from within Jupiter's magnetosphere using various magnetic field models.

This is part of a community code project :

Magnetospheres of the Outer Planets Group Community Code

This module forms part of the JupiterMag package for Python.

3.6.1 Cloning and Building

This module requires a few submodules to be fetched, so the following command should clone everything:

```
git clone --recurse-sumodules https://github.com/mattkjames7/libjupitermag.git
```

This library requires `g++`, `make`, and `ld` (Linux) or `libtool` (Mac) in order to be compiled. On Windows, these tools can be provided by TDM-GCC.

To build in Linux and Mac, simply run

```
cd libjupitermag
make
#optionally install the library
sudo make install
```

where the installation defaults to `/usr/local` but can be changed using the `PREFIX` argument, e.g.:

```
sudo make install PREFIX=/usr
```

It can also be uninstalled:

```
make uninstall
```

Under Windows powershell/command line:

```
cd libjupitermag
compile.bat
```

or under Linux but building for Windows:

```
cd libjupitermag
make windows
```

After a successful build, a new library (`libjupitermag.so`) or DLL (`libjupitermag.dll`) should appear in the `lib/libjupitermag` directory.

3.6.2 Usage

The shared object library created by compiling this project should be accessible by various programming languages. This section mostly covers C++, other languages should be able to use the functions defined in the `extern "C"` section of the header file quite easily. For Python, it is straightforward to use `ctypes` to access this library, similarly, IDL could use the `CALL_EXTERNAL` function.

Linking to the library

Here is a very basic example of how to link to the code using C++ and print the version of the library:

```
/* test.cc */
#include <stdio.h>
#include <jupitermag.h>

int main() {
        /* simply print the version of the library */
        printf("libjupitermag version: $%d.%d.%d\n",
                        LIBJUPITERMAG_VERSION_MAJOR,
                        LIBJUPITERMAG_VERSION_MINOR,
                        LIBJUPITERMAG_VERSION_PATCH);
        return 0;
}
```

If the library was installed using `sudo make install`, then the library can be linked to during compiling time with:

```
g++ test.c -o test -ljupitermag
```

Otherwise, the header should be included using a relative or absolute path, e.g:

```
/* instead of this */
#include <jupitermag>
/* use something like this */
#include "include/jupitermag.h"
```

then compile, e.g.:

```
# absolute path
g++ test.cc -o test -L:/path/to/libjupitermag.so

# or relative path
g++ test.cc -o test -Wl,-rpath='$$ORIGIN/../lib/libjupitermag' -L ..lib/libjupitermag -ljupitermag
```

Calling Field Models

Internal field models can be called using the `InternalModel` object, e.g.:

```
#include <stdio.h>
#include <jupitermag.h>

int main () {
        /* create an instance of the object */
        InternalModel modelobj = InternalModel();

        /* set the model to use */
        modelobj.SetModel("jrm09");

        /* get the model vectors at some position */
        double x = 10.0, y = 0.0, z = 0.0;
        double Bx, By, Bz;
        modelobj.Field(x,y,z,&Bx,&By,&Bz);

        printf("B at [%f,%f,%f] = [%f,%f,%f]\n",x,y,z,Bx,By,Bz);
}
```

There are also simple functions for each of the models included in the library, see the table below.

Model	C String	Field Function	Reference
JRM33	`jrm33`	`jrm33Field`	Connerney et al., 2022
JRM09	`jrm09`	`jrm09Field`	Connerney et al., 2018
ISaAC	`isaac`	`isaacField`	Hess et al., 2017
VIPAL	`vipal`	`vipalField`	Hess et al., 2011
VIP4	`vip4`	`vip4Field`	Connerney 2007
VIT4	`vit4`	`vit4Field`	Connerney 2007
O4	`o4`	`o4Field`	Connerney 1981
O6	`o6`	`o6Field`	Connerney 2007
GSFC15evs	`gsfc15evs`	`gsfc15evsField`	Connerney 1981
GSFC15ev	`gsfc15ev`	`gsfc15evField`	Connerney 1981
GSFC13ev	`gsfc13ev`	`gsfc13evField`	Connerney 1981
Ulysses 17ev	`u17ev`	`u17evField`	Connerney 2007
SHA	`sha`	`shaField`	Connerney 2007
Voyager 1 17ev	`v117ev`	`v117evField`	Connerney 2007
JPL15ev	`jpl15ev`	`jpl15evField`	Connerney 1981
JPL15evs	`jpl15evs`	`jpl15evsField`	Connerney 1981
P11A	`p11a`	`p11aField`	
External Model			
Con 2020	`con2020`	`Con2020Field`	Connerney et al., 1981; Edwards et al., 2001; Connerney et al., 2020

Table 3.1: Internal and External Field Models

Field Tracing

Field tracing can be done using the `Trace` object, e.g.:

```
#include <stdio.h>
#include <jupitermag.h>
#include <vector>
```

```
int main () {
        /* set initial position to start trace from (this can be an array
                for multiple traces) */
        int n = 1;
        double x0 = 5.0;
        double y0 = 0.0;
        double z0 = 0.0;
        int nalpha = 1;
        double alpha = 0.0;

        printf("Create field function vector\n");
        /* store the function pointers for the components of the
        model to be included in the trace */
        std::vector<FieldFuncPtr> Funcs;

        /* internal model */
        Funcs.push_back(jrm09Field);

        /* external model */
        Funcs.push_back(Con2020Field);

        /* initialise the trace object */
        printf("Create Trace object\n");
        Trace T(Funcs);

        /* add the starting positions for the traces */
        printf("Add starting position\n");
        T.InputPos(n,&x0,&y0,&z0);

        /* configure the trace parameters, leaving this empty will
        use default values for things like minimum and maximum step size */
        printf("Set the trace parameters \n");
        T.SetTraceCFG();

        /* set up the alpha calculation - the angles (in degrees) of each
        polarization angle. This is generally used for ULF waves */
        printf("Initialize alpha\n");
        T.SetAlpha(nalpha,&alpha);

        /* Trace */
        printf("Trace\n");
        T.TraceField();

        /* trace distance, footprints, Rnorm */
        printf("Footprints etc...\n");
        T.CalculateTraceDist();
        T.CalculateTraceFP();
        T.CalculateTraceRnorm();

        /* calculate halpha for each of the polarization angles
           specified above*/
        printf("H_alpha\n");
        T.CalculateHalpha();
}
```

The above code traces along the magnetic field using the JRM09 internal and Con2020 external field models together. The trace coordinates and field vectors at each step can be obtained from the member variables `T.x_`, `T.y_`, `T.z_`, `T.Bx_`, `T.By_`, and `T.Bz_`, where each is a 2D array with the shape (`T.n_`,`T.MaxLen_`), where `T.n_` is the number of traces and `T.MaxLen_` is the maximum number of steps allowed in the trace. The number of steps taken in each trace is defined in the `T.nstep_` array.

3.7 con2020: Python implementation of Jupiter's magnetodisc model

3.8 con2020

Python implementation of the Connerney et al., 1981 and Connerney et al., 2020 Jovian magnetodisc model. This model provides the magnetic field due to a "washer-shaped" current near to Jupiter's magnetic equator. This model code uses either analytical equations from Edwards et al., 2001 or the numerical integration of the Connerney et al., 1981 equations to provide the magnetodisc field, depending upon proximity to the disc along z and the inner edge of the disc, r_0.

For the IDL implementation of this model, see: https://github.com/marissav06/con2020$_{i}dl$

Or for Matlab: https://github.com/marissav06/con2020$_{m}atlab$

A PDF documentation file is available here: con2020$_{f}inal_{c}ode_{d}ocumentation_{j}une9_{2}022.pdf$. It describes the Connerney current sheet model and general code development (equations used, numerical integration assumptions, accuracy testing, etc.). Details specific to the Python code are provided in this readme file.

These codes were developed by Fran Bagenal, Marty Brennan, Matt James, Gabby Provan, Marissa Vogt, and Rob Wilson, with thanks to Jack Connerney and Masafumi Imai. They are intended for use by the Juno science team and other members of the planetary magnetospheres community. Our contact information is in the documentation PDF file.

3.8.1 Installation

Install the module using `pip3`:

```
pip3 install --user con2020

#or if you have previously installed using this method
pip3 install --upgrade --user con2020
```

Or using this repo:

```
#clone the repo
git clone https://github.com/gabbyprovan/con2020
cd con2020

#EITHER create a wheel and install (X.X.X is the current version number)
python3 setup.py bdist_wheel
pip3 install --user dist/con2020-X.X.X-py3-none-any.whl

#or directly install using setup.py
python3 setup.py insall --user
```

3.8.2 Usage

To call the model, an object must be created first using `con2020.Model()`, where the default model parameters, model equations used or coordinate systems of input and output can be altered using keywords, e.g:

```python
import con2020

#initialize a model object with default parameters
def_model = con2020.Model()

#initialize a model which uses spherical polar coordinates for input and output
sph_model = con2020.Model(CartesianIn=False,CartesianOut=False)

#initialize a model with custom parameters (longhand)
cust_model0 = con2020.Model(mu_i_div2__current_parameter_nT=150.0,
                                                        r0__inner_rj=9.5,
                                                     d__cs_half_thickness_rj=3.1)

#equivalently, a custom parameter model (shorthand)
cust_model1 = con2020.Model(mu_i=150.0,r0=9.5,d=3.1)
```

Once a model object is initialized, the model field can be obtained by calling the member function `Field()` and supplying input coordinates as three scalars, or three arrays (all of which are in right-handed System III), e.g.:

```
#Example 1: the model at a single Cartesian position (all in Rj)
x = 5.0
y = 10.0
z = 6.0
Bcart = def_model.Field(x,y,z)
#Result:
Bxyz=[15.57977074, 36.88229249, 63.02051163] nT
#Calculated using the default con2020 model keywords and the hybrid approximation.

#Example 2: the model at an array of positions of spherical polar coordinates
r = np.array([10.0,20.0])                                  #radial distance in Rj
theta = np.array([30.0,35.0])*np.pi/180.0        #colatitude in radians
phi = np.array([90.0,95.0])*np.pi/180.0        #east longitude in radians
Bpol = sph_model.Field(r,theta,phi)
#Result:
Spherical polar Brtp =[63.32354453 ,31.15790459], [-21.01051861 , -6.86773727], [-3.61151705, -2.726260
Cartesian       Bxyz =[3.61151705, 1.6486016], [13.4661294,  12.43672946], [65.34505753, 29.46223351] n
#Calculated using the default con2020 model keywords and the hybrid approximation.
```

The output will be a `numpy.ndarray` with a shape `(n,3)`, where `n` is the number of input coordinates, `B[:,0]` corresponds to either `Bx` or `Br`; `B[:,1]` corresponds to `By` or `Btheta`; and `B[:,2]` corresponds to either `Bz` or `Bphi`. A full list of model keywords is shown below:

Keyword (long)	Keyword (short)	Default Value	Description
mu_i_div2__current_parameter_nT	mu_i	139.6*	Current sheet current density
i_rho__radial_current_MA	i_rho	16.7*	†Radial current intensity in M from Connerney et al 2020.
r0__inner_rj	r0	7.8	Inner edge of the current sheet
r1__outer_rj	r1	51.4	Outer edge of the current shee
d__cs_half_thickness_rj	d	3.6	Current sheet half thickness in
xt__cs_tilt_degs	xt	9.3	Tilt angle of the current sheet
xp__cs_rhs_azimuthal_angle_of_tilt_degs	xp	155.8	(Right-Handed) Longitude tow
equation_type		'hybrid'	Which method to use, can be: 'analytic' - use only the ana 'integral' - numerically inte 'hybrid' - a combination of a
error_check		True	Check errors on inputs the the
CartesianIn		True	If True (default) then the inpu
CartesianOut		True	If True the output magnetic fi
azfunc		'connerney'	Which model to use for the az
DeltaRho		1.0	Scale length over which smoot
DeltaZ		0.1	Scale length over which smoot
g		417659.3836476442	§Magnetic dipole parameter, n
w0_open		0.1	§Ratio of plasma to Jupiter's a
w0_om		0.35	§Ratio of plasma to Jupiter's a
thetamm		16.1	§Colatitude of the centre of th
dthetamm		0.5	§Colatitude range over which t
thetaoc		10.716	§Colatitude of the centre of th
dthetaoc		0.125	§Colatitude range of the open-

*Default current densities used here are averages provided in Connerney et al., 2020 (see Figure 6), but can vary from one pass to the next. Table 2 of Connerney et al., 2020 provides a list of both current densities for 23 out of the first 24 perijoves of Juno.

† This is only applicable for the Connerney et al., 2020 model for B_ϕ.

§ These parameters are used to configure the L-MIC model for B_ϕ.

The `con2020.Test()` function should produce figure 3.4:

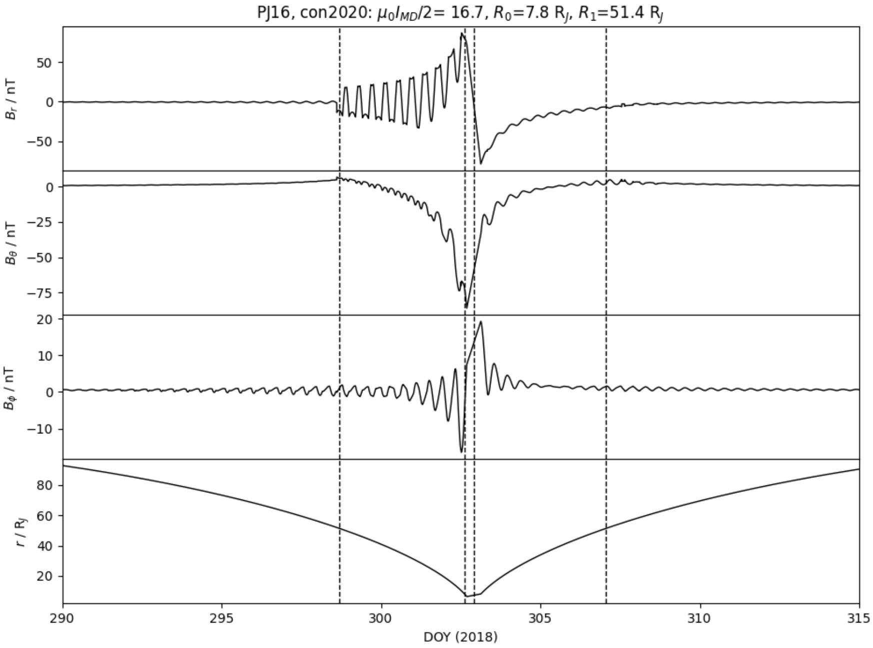


Figure 3.4: Test of `con2020` code on a Juno orbit.

3.9 `libcon2020`: C++ implementation of Jupiter's magnetodisc model

C++ implementation of the Connerney et al., 1981 and Connerney et al., 2020 Jovian magnetodisc model. This model provides the magnetic field due to a "washer-shaped" current disc near to Jupiter's magnetic equator. The model uses the analytical equations from Edwards et al., 2001 or Connerney et al., 1981; or the numerical integration of the Connerney et al., 1981 equations to provide the magnetic field vectors due to the azimuthal current. This code also implements the Connerney et al., 2020 radial current and the Leicester magnetosphere-ionosphere coupling (L-MIC, Cowley et al., 2005, 2008) models which provide the azimuthal component of the magnetic field.

This is part of `libjupitermag`, which is part of a greater effort to provide community code for the Jovian magnetosphere:

`Magnetospheres of the Outer Planets Group Community Code`

3.9.1 Building libcon2020

To build this library in Linux or Mac OS:

```
#clone this repo
git clone https://github.com/mattkjames7/libcon2020.git
cd libcon2020

#build
make

#optionally install it system wide
sudo make install
```

In Windows:

```
git clone https://github.com/mattkjames7/libcon2020.git
cd libcon2020

.\compile.bat
```

With a system wide installation, the compiler and linker will be able to locate the library and its header, otherwise absolute paths must be provided for linking and including. In Windows, there is an experimental script `install.bat` which will copy the DLL and headers to folders within the `C:\TDC-GCC-64\` directory. This is experimental; instead, it might be better to copy the headers and the DLL to the root directory of the executable linked to it.

Uninstallation can be achieved in Linux and Mac using `sudo make uninstall`.

3.9.2 Usage

Linking to and Including libcon2020

If a system-wide installation is successful, then the library may be linked to simply by including the `-lcon2020` flag while compiling/linking. Otherwise, the path to the library must also be included, e.g. `-L /path/to/lib/directory -lcon2020`. In Windows, the DLL should be placed in the root directory of the executable linked to it.

This library includes a header file `include/con2020.h` which is compatible with both C and C++. This header contains the full set of function and class prototypes for use with C++; it also includes C-compatible wrapper functions. The wrapper functions in the header file would also provide the easiest ways to link other languages to the library such as Python, IDL, and Fortran.

If the library was installed system-wide, then the headers may be included using `#include <con2020.h>`. Otherwise, a relative or absolute path to the headers must be used, e.g. `#include "path/to/con2020.h"`.

C++ usage

This section briefly describes some C++ specific examples for using the `libcon2020` library, while the following section is also somewhat applicable.

Access the model using the `Con2020` class:

```
/* contents of cppexample.cc */
#include <stdio.h>
#include <con2020.h>

int main () {
        /* create an instance of the model */
        Con2020 model;

        /* set coordinate system to be Cartesian SIII (default) */
        model.SetCartIn(true);
        model.SetCartOut(true);

        /* create some variables to store a field vector in,
         * note that positions are in units of Rj */
        double x = 11.0;
        double y = 5.0;
        double z = -10.0;
        double Bx, By, Bz;

        /* call the model */
        model.Field(x,y,z,&Bx,&By,&Bz);
        printf("B=[%5.1f,%5.1f,%5.1f] nT at [%4.1f,%4.1f,%4.1f] Rj\n",Bx,By,Bz,x,y,z);

        /* alternatively obtain an array of field vectors in spherical polar coords */
        model.SetCartIn(false);
        model.SetCartOut(false);
        double r[] = {5.0,10.0,15.0};
        double theta[] = {1.0,1.5,2.0};
        double phi[] = {0.0,0.1,0.2};
        double Br[3], Bt[3], Bp[3];

        model.Field(3,r,theta,phi,Br,Bt,Bp);
        int i;
        for (i=0;i<3;i++) {
                printf("B=[%5.1f,%5.1f,%5.1f] nT at r = %4.1f Rj, theta = %4.1f rad, phi = %4.1f rad\n"
        }
```

```
        return 0;
}
```

In the example above, the model can be configured by using class member functions (e.g., `Con2020::SetCartIn()`), see the table below for a full list of configurable parameters. The `Con2020::Field()` function is overloaded, so it can either accept a single input position to provide a single field vector or can accept an array of positions to provide an array of field vectors. Compile and run the above example using

```
g++ cppexample.cc -o cppexample -lcon2020
./cppexample
```

Other Languages

In other languages, it is easier to run the model code by using the wrapper functions listed in both `con2020.h`. For example, in C:

```
/* contents of cexample.c */
#include <stdio.h>
#include <stdbool.h>
#include <con2020.h>

int main() {

        /* we can obtain a single field vector like this,
         * using the default model parameters */
        double x = 10.0;
        double y = 20.0;
        double z = 5.0;
        double Bx, By, Bz;
        Con2020Field(x,y,z,&Bx,&By,&Bz);
        printf("B=[%5.1f,%5.1f,%5.1f] nT at [%4.1f,%4.1f,%4.1f] Rj\n",Bx,By,Bz,x,y,z);

        /* or using arrays */
        double xa[] = {10.0,20.0,30.0};
        double ya[] = {5.0,10.0,15.0};
        double za[] = {10.0,10.0,10.0};
        double Bxa[3], Bya[3], Bza[3];
        Con2020FieldArray(3,xa,ya,za,Bxa,Bya,Bza);
        int i;
        for (i=0;i<3;i++) {
                printf("B=[%5.1f,%5.1f,%5.1f] nT at [%4.1f,%4.1f,%4.1f] Rj\n",
                                        Bxa[i],Bya[i],Bza[i],xa[i],ya[i],za[i]);
        }

        /* we can retrieve the current model parameters */
        double mui, irho, r0, r1, d, xt, xp, DeltaRho, DeltaZ, g, w0_open,
                        w0_om, thetamm, dthetamm, thetaoc, dthetaoc;
        bool Edwards, ErrChk, CartIn, CartOut, smooth;
        char eqtype[16];
        char azfunc[16];
        GetCon2020Params(&mui,&irho,&r0,&r1,&d,&xt,&xp,eqtype,&Edwards,&ErrChk,
                                        &CartIn,&CartOut,&smooth,&DeltaRho,&DeltaZ,&g,azfunc,
                                        &w0_open,&w0_om,&thetamm,&dthetamm,&thetaoc,&dthetaoc);

        /* these parameters may be edited and passed back to the model */
        irho = 0.0;
        strcpy(eqtype,"analytic");
        SetCon2020Params(mui,irho,r0,r1,d,xt,xp,eqtype,Edwards,ErrChk,
                                        CartIn,CartOut,smooth,DeltaRho,DeltaZ,g,azfunc,
                                        w0_open,w0_om,thetamm,dthetamm,thetaoc,dthetaoc);
}
```

Compile and run using:

```
gcc cexample.c -o cexample -lcon2020
./cexample
```

3.9.3 Model Parameters

This table lists all of the configurable parameters currently included in the code.

Parameter	Default	Con2020 Member Functions		Description
mui	139.6	SetAzCurrentParameter()	GetAzCurrentParameter()	Azimuthal current she eter, nT.
irho	16.7	SetRadCurrentParameter()	GetRadCurrentParameter()	Radial current intensit
r0	7.8	SetR0()	GetR0()	Inner edge of the curr R_J.
r1	51.4	SetR1()	GetR1()	Outer edge of the curr R_J.
d	3.6	SetCSHalfThickness()	GetSCHalfThickness()	Half thickness of th sheet, R_J.
xt	9.3	SetCSTilt()	GetCSTilt()	Tilt angle of the cur away from the SIII z-a
xp	155.8	SetCSTiltAzimuth()	GetCSTiltAzimuth()	Right-handed longituc which the current shee °.
eqtype	"hybrid"	SetEqType()	GetEqType()	Set what method to u "analytic" - use anal tions "integral" - use fu equations "hybrid" - use a comb both
Edwards	true	SetEdwardsEqs()	GetEdwardsEqs()	Switch between the E al., 2001 and Conner 1981 analytical equati
ErrChk	true	SetErrCheck()	GetErrChk()	Check for errors on the Con2020::Field(), se for a slightly risky spe
CartIn	true	SetCartIn()	GetCartIn()	If true, then input c will be assumed to be sian (in units of R_J), spherical polar coordin units of R_J; θ and ϕ in
CartOut	true	SetCartOut()	GetCartOut()	If true, then output fi will be in Cartesian S nates, otherwise they ented such that the thr nents are radial, merid azimuthal.
smooth	true	SetSmooth()	GetSmooth()	Use Stan's tanh-based to smooth over the boundaries in the ρ dir across $\pm d$ in the z dire
DeltaRho	1.0	SetDeltaRho()	GetDeltaRho()	Scale length over whic ing is done in the ρ dir
DeltaZ	0.1	SetDeltaZ()	GetDeltaZ()	Scale length over whic ing is done in the z di
g	417659.3836476442	SetG()	GetG()	Magnetic dipole paran
azfunc	"connerney"	SetAzimuthalFunc()	GetAzimuthalFunc()	Set the method to use late the azimuthal fie the radial component rent: "connerney" - use th ney et al., 2020 azim model "lmic" - use the L-M (Connerney et al., 200
w0_open	0.1	SetOmegaOpen()	GetOmegaOpen()	Ratio of plasma to Ju gular velocity on open
w0_om	0.35	SetOmegaOM()	GetOmegaOM()	Ratio of plasma to Ju gular velocity in the c netosphere.
thetamm	16.1	SetThetaMM()	GetThetaMM()	Colatitude of the cen middle magnetospher the plasma transitions tating to sub-corotatin

3.10 jrm33: The JRM33 model in Python

3.10.1 Installation

Install using pip:

```
pip3 install jrm33 --user
```

Or by cloning this repo:

```
git clone https://github.com/mattkjames7/jrm33.git
cd jrm09

# EITHER create a wheel and install (replace X.X.X with the version number):
python3 setup.py bdist_wheel
pip3 install dist/jrm33-X.X.X-py3-none-any.whl --user

% OR install directly using setup.py
python3 setup.py install --user
```

3.10.2 Usage

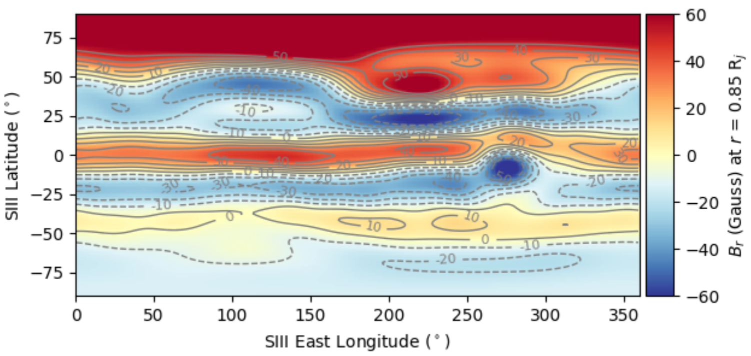
The model accepts right-handed System III coordinates either in Cartesian form (`jrm33.ModelCart()`) or in spherical polar form (`jrm33.Model()`), e.g.:

```
import jrm33

# get some Cartesian field vectors (Deg keyword is optional)
Bx, By, Bz = jrm33.ModelCart(x, y, z, Deg=13)

# or spherical polar ones
Br, Bt, Bp = jrm33.Model(r, theta, phi, Deg=13)
```

Please read the docstrings for `jrm33.Model()` and `jrm33.ModelCart()` using `help` or `?` e.g. `help(jrm33.Model)`. There is also a test function which requires `matplotlib` to be installed:

```
# evaluate the model at some R
jrm33.Test(R=0.85)
```

which produces this (based on figure 4 of Connerney et al. 2018):



3.11 jrm09: The JRM09 model in Python

3.12 Installation

Install using pip:

```
pip3 install jrm09 --user
```

Or by cloning this repo:

```
git clone https://github.com/mattkjames7/jrm09.git
cd jrm09

# EITHER create a wheel and install (replace X.X.X with the version number):
python3 setup.py bdist_wheel
pip3 install dist/jrm09-X.X.X-py3-none-any.whl --user

# OR install directly using setup.py
python3 setup.py install --user
```

3.13 Installation

Install using pip:

```
pip3 install jrm09 --user
```

Or by cloning this repo:

```
git clone https://github.com/mattkjames7/jrm09.git
cd jrm09

% EITHER create a wheel and install (replace X.X.X with the version number):
python3 setup.py bdist_wheel
pip3 install dist/jrm09-X.X.X-py3-none-any.whl --user

% OR install directly using setup.py
python3 setup.py install --user
```

3.14 Usage

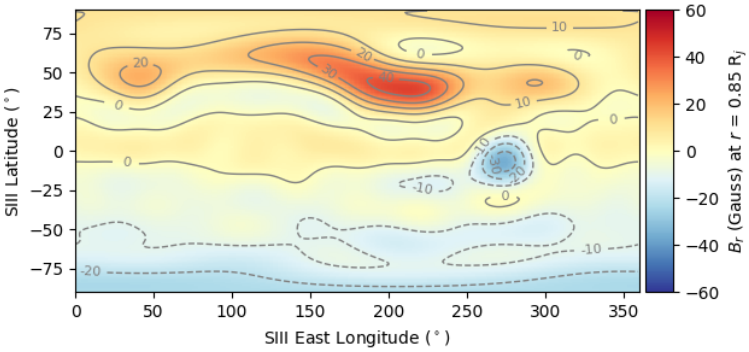
The model accepts right-handed System III coordinates either in Cartesian form (`jrm09.ModelCart()`) or in spherical polar form (`jrm09.Model()`), e.g.:

```
import jrm09

% get some Cartesian field vectors (MaxDeg keyword is optional)
Bx, By, Bz = jrm09.ModelCart(x, y, z, MaxDeg=10)

% or spherical polar ones
Br, Bt, Bp = jrm09.Model(r, theta, phi, MaxDeg=10)
```

Please read the docstrings for `jrm09.Model()` and `jrm09.ModelCart()` using `help` or `?` e.g. `help(jrm09.Model)`. There is also a test function which requires `matplotlib` to be installed:

```
#evaluate the model at some R
jrm09.Test(R=0.85)
```

which produces this (based on figure 4 of Connerney et al. 2018):



3.15 `vip4model`: The VIP4 model in Python

3.16 Installation

Install using `pip`:

```
pip3 install vip4model --user
```

Or by cloning this repo:

```
git clone https://github.com/mattkjames7/vip4model.git
cd vip4model

# EITHER create a wheel and install (replace X.X.X with the version number):
python3 setup.py bdist_wheel
pip3 install dist/vip4model-X.X.X-py3-none-any.whl --user

# OR install directly using setup.py
python3 setup.py install --user
```

3.17 Usage

The model accepts right-handed System III coordinates either in Cartesian form (`vip4model.ModelCart()`) or in spherical polar form (`vip4model.Model()`), e.g.:

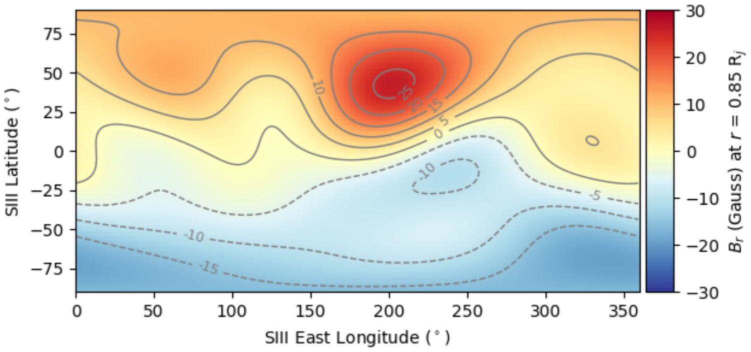
```
import vip4model

# get some Cartesian field vectors (MaxDeg keyword is optional)
Bx, By, Bz = vip4model.ModelCart(x, y, z, MaxDeg=4)

# or spherical polar ones
Br, Bt, Bp = vip4model.Model(r, theta, phi, MaxDeg=4)
```

Please read the docstrings for `vip4model.Model()` and `vip4model.ModelCart()` using `help` or `?` e.g. `help(vip4model.Model)`.

There is also a test function which requires `matplotlib` to be installed:

```
# evaluate the model at some R
vip4model.Test(R=0.85)
```

which produces this (based on figure 4 of Connerney et al. 2018):



SIII Latitude (°)
SIII East Longitude (°)
B_r (Gauss) at r = 0.85 R_j

Chapter 4

Spacecraft Data

4.1 Arase: Download and read Arase data

4.1.1 Installation

Install from PyPI:

```
pip3 install Arase --user
```

or

```
python3 -m pip install Arase --user
```

Set the ARASE_PATH variable by placing the following at the end of ~/.bashrc:

```
export ARASE_PATH=/path/to/arase/data
```

4.1.2 Downloading Data

Most instrument data can be downloaded using the DownloadData function contained in the instruments submodule, e.g.:

```
Arase.XXX.DownloadData(L, prod, Date=Date, Overwrite=Overwrite)
```

where XXX can be replaced with the instrument names: HEP, LEPe, LEPi, MEPe, MEPi, MGF or XEP. L is an integer and prod is a string which correspond to the level and data product provided by the instrument, respectively (see the table in "Current Progress"). Date determines the range of dates to download data for. The Date keyword can be a single date, a list of specific dates to download, or a 2 element list defining the start and end dates (by default Date = [20170101, 20200101]). Overwrite will force the routine to overwrite existing data.

This method will work for PWE data:

```
Arase.PWE.DownloadData(subcomp, L, prod, Date=Date, Overwrite=Overwrite)
```

where subcomp is the sub-component of the instrument (see table below).

To download the position data:

```
Arase.Pos.DownloadData(prod, Date=Date, Overwrite=Overwrite)
```

where prod is either 'l3' or 'def'. The 'def' option is needed for position-related functions elsewhere in the Arase module.

4.1.3 Position and Tracing

1. Download position data:

   ```
   Arase.Pos.DownloadData('def')
   ```

2. Convert to a binary format (this allows for quicker reading):

   ```
   Arase.Pos.ConvertPos()
   ```

3. Save field traces:

```
Arase.Pos.SaveFieldTraces(Model=Model, StartDate=StartDate, EndDate=EndDate)
```

   where `Model` is either `'T89'`, `'T96'`, `'T01'` or `'TS05'` (`'T96'` by default). `StartDate` and `EndDate` are the start and end dates to perform traces for, both are integers of the format `yyyymmdd`.

4. To read the position data:

```
pos = Arase.Pos.GetPos()
```

5. To read the traces:

```
tr = Arase.Pos.ReadFieldTraces(Date)
```

4.1.4 Reading Data

MGF Data

```
data = Arase.MGF.ReadMGF(Date)
```

This returns a `numpy.recarray` object which contains the time-series data. The `Date` argument may be a single date, a list of dates, or a 2 element `list` of dates defining the start and end date to load.

Particle Omni-directional Spectra

```
data = Arase.LEPe.ReadOmni(Date)
```

For other instruments, replace `LEPe` with one of the following: `LEPi`, `MEPe`, `MEPi`, `HEP` or `XEP`. `data` is a dictionary which will contain dates, times, energy bins, and instances of the `Arase.Tools.PSpecCls` object. The `PSpecCls` object contains all of the spectral information stored within it and is usually identified by the dictionary key containing `'Flux'`. The `PSpecCls` object has an in-built method for plotting the spectrograms, e.g.:

```
data['eFlux'].Plot()
```

will plot the electron flux spectrogram from the LEPe data loaded above. To list the keys of a dictionary, use `list(data.keys())`.

Combined Particle Spectra

Two functions are available which will load the data for multiple instruments at the same time.

For electrons:

```
E = Arase.Electrons.ReadOmni(Date)
```

and for ions:

```
H, He, O = Arase.Ions.ReadOmni(Date)
```

where `E`, `H`, `He`, and `O` are all instances of `SpecCls`.

Single Spectra

The `SpecCls` object has the ability to return single spectra, e.g.:

```
import Arase
import matplotlib.pyplot as plt

# read in the electrons - this should work with any SpecCls object
spec = Arase.Electrons.ReadOmni(Date)

# for the energy bins and particle flux data
e, dJdE, _ = spec.GetSpectrum(Date, ut)

# for velocity and phase space density
v, f, _ = spec.GetSpectrum(Date, ut, xparam='V', yparam='PSD')
```

```
# or to plot
plt.figure(figsize=(8, 4))
ax0 = spec.PlotSpectrum(Date, ut, xparam='E', yparam='Flux', Split=True, fig=plt, maps=[2, 1, 0
ax1 = spec.PlotSpectrum(Date, ut, xparam='V', yparam='PSD', Split=True, fig=plt, maps=[2, 1, 1,
plt.tight_layout()

# for more information, read the docstrings:
spec.GetSpectrum?
spec.PlotSpectrum?
```

3D Particle Spectra

These data are not currently placed into an object like `PSpecCls`. For instruments which provide 3D spectra, there is a function `Read3D` which will simply read the CDF file for a given date and list all of the data and corresponding metadata into two dictionaries, e.g.:

```
data, meta = Arase.LEPe.Read3D(Date)
```

Pitch Angle Distributions

For particle instruments with 3D flux data, there is a method to convert these to pitch angle distributions (PADs). The PADs are calculated using the MGF data and the elevation/azimuth angles of the instruments in GSE coordinates where provided in the level 2 `3dflux` data products. It was possible to compare this method to the angles provided by the level 3 `3dflux` product from the MEPe instrument, and almost all pitch angles were within about 1-2 degrees. **WARNING: these data should be used with caution - they may not be correct.**

To store the PADs:

```
import Arase
Arase.LEPe.SavePADs(Date, na=18, Overwrite=False, DownloadMissingData=True, DeleteNewData=True,
```

The above code will bin up the 3D LEPe fluxes from a single date into `na` pitch angle bins (always in the range 0 to 180 degrees). The `Overwrite` keyword will force the overwriting of previously created PAD files. `DownloadMissingData` will download any missing `3dflux` data and MGF data. `DeleteNewData` will delete the newly downloaded `3dflux` data after creating the PAD data because some of the `3dflux` files are ¿ 500 MB.

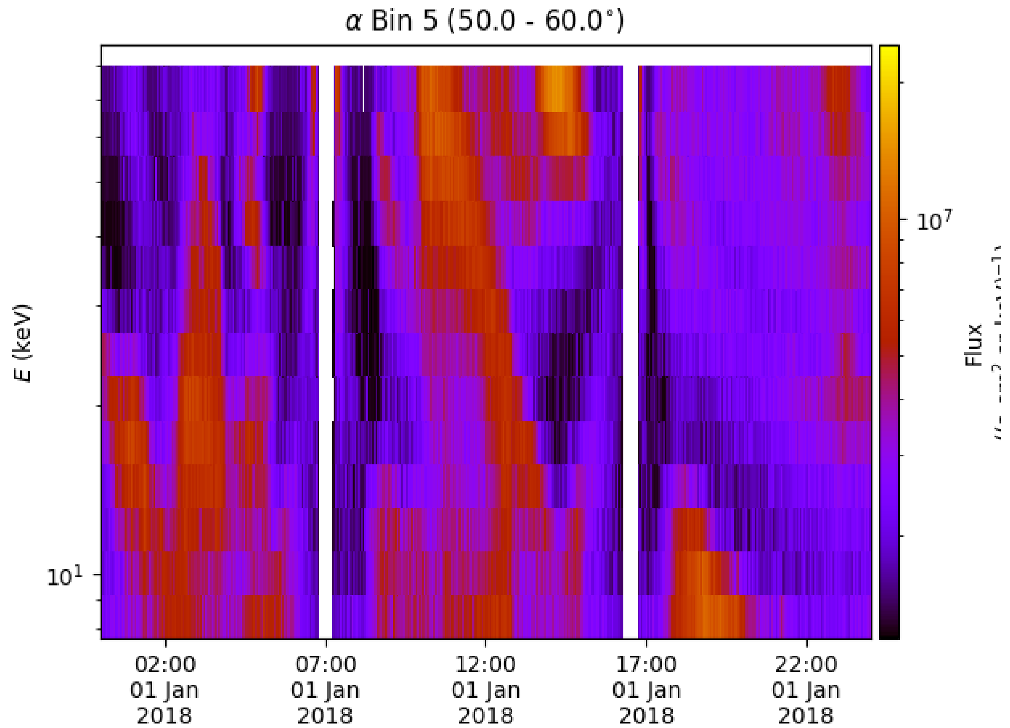
To read PADs:

```
pad = Arase.LEPe.ReadPAD(Date, SpecType, ReturnSpecObject=True)
```

This will read the PAD spectra from a single date for a given `SpecType` (e.g. `'eFlux'` or `'H+Flux'`, depending on the instrument). The returned object will either be a `dict` containing just the data if `ReturnSpecObject=False`, or a `Arase.Tools.PSpecPADCls` object if `ReturnSpecObject=True`. The `PSpecPADCls` object allows the plotting of spectrograms, 1D spectra, and 2D spectra.

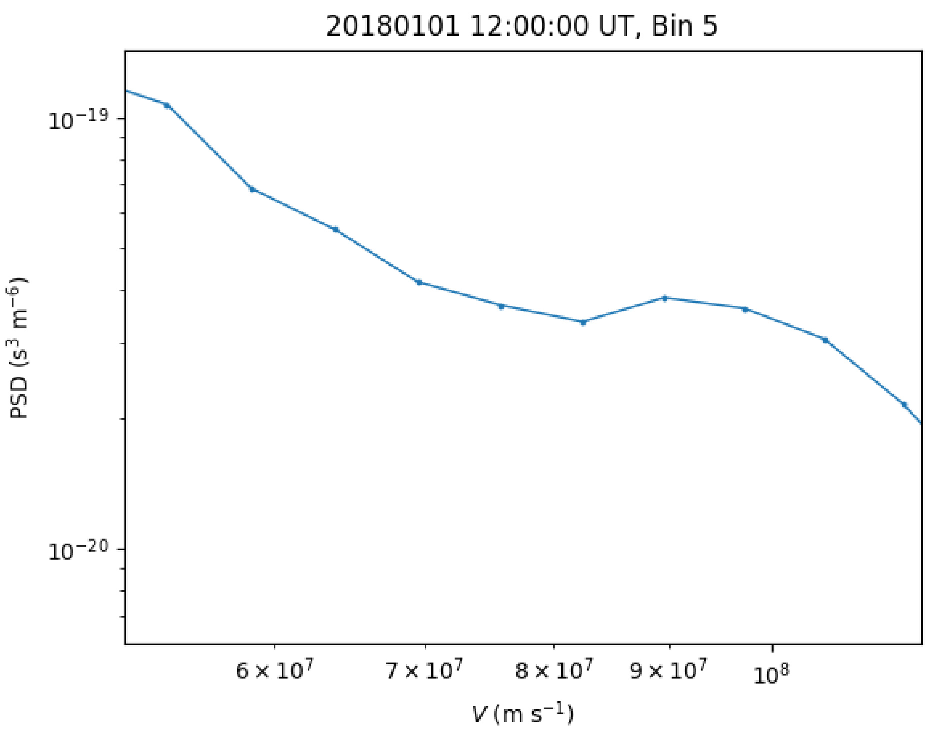
For a spectrogram of a specific pitch angle bin:

```
pad = Arase.MEPe.ReadPAD(20180101, 'eFlux')
pad.PlotSpectrogram(Bin=5)
```



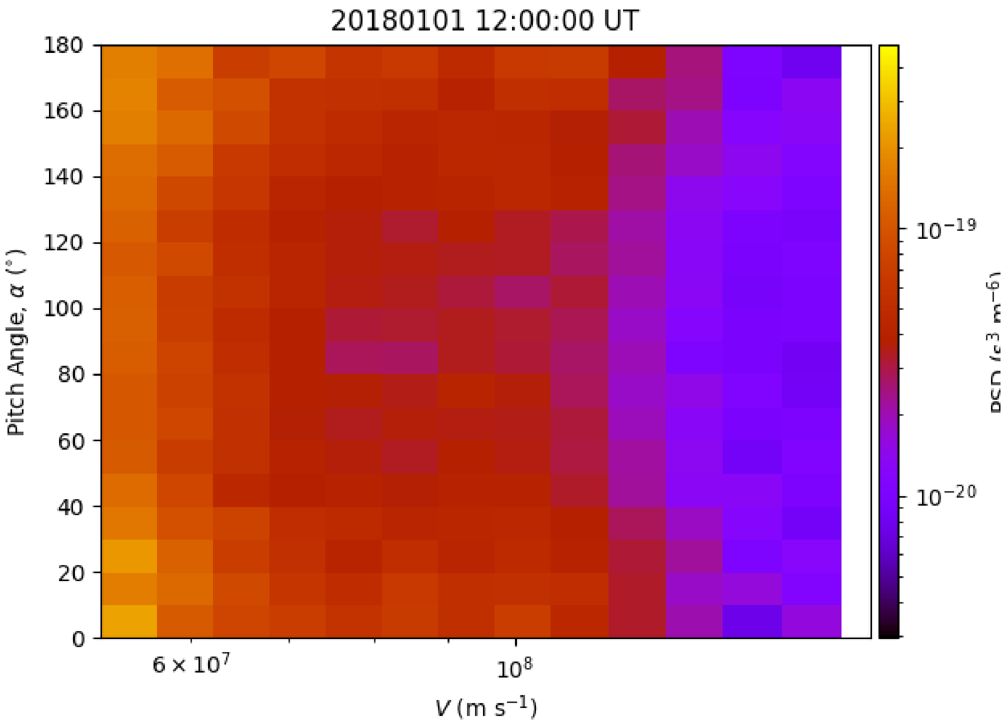
Or a 1D spectrum:

```
pad.PlotSpectrum1D(12.0, Bin=5, xparam='V', yparam='PSD')
```



Or a 2D spectrum:

```
pad.PlotSpectrum2D(12.0, xparam='V', zparam='PSD')
```



4.1.5 Current Progress

Instrument	Subcomponent	Level	Product	Download	Read	Plot
HEP		2	omniflux			
LEPe		2	omniflux			
LEPe		2	3dflux		×	×
LEPi		2	omniflux			
LEPi		2	3dflux		×	×
MEPe		2	omniflux			
MEPe		2	3dflux		×	×
MEPe		3	3dflux		×	×
MEPi		2	omniflux			
MEPi		2	3dflux		×	
MEPi		3	3dflux			×
MGF		2	8sec			×
PWE	efd	2	spec			
PWE	hfa	2	high			
PWE	hfa	2	low			
PWE	hfa	3				×
PWE	ofa	2	complex	×	×	×
PWE	ofa	2	matrix	×	×	×
PWE	ofa	2	spec		×	×
XEP		2	omniflux			

- - Works
- × - Not working yet. In the case of 3D data, a `SpecCls3D` object needs to be written. For MGF and level 3 hfa data, it's a simple case of plotting a line.
- - Probably works, but have no access to the data to be able to test it.
- × - Currently, 3D spectra can only be read into dictionaries as a `SpecCls3D` object is needed.

4.2 `RBSP`: Download and read Van Allen Probe data

Some tools for loading RBSP (Van Allen Probe) data

4.3 `cluster`: Download and read Cluster data

Download and read in data from the Cluster mission.

4.4 `pyCRRES`: Download and read CRRES data

Code for the Combined Release and Radiation Effects Satellite (CRRES).

4.5 `themissc`: Download and read THEMIS data

A python package to download and read THEMIS spacecraft data.

4.6 `imageeuv`: Download and read IMAGE EUV data

4.7 `imagerpi`: Download and read IMAGE RPI data

4.8 `imagePP`: Download and read Goldstein's plasmapause dataset

Simple Python code to download and use Jerry Goldstein's plasmapause database.

4.9 Installation

Clone the project and build:

```
git clone https://github.com/mattkjames7/ImagePP.git
cd ImagePP

#build the wheel file
python3 setup.py bdist_wheel

#install using pip (replace x.x.x with current version)
pip3 install --user dist/ImagePP-x.x.x-py3-none-any.whl
```

The `ImagePP` module requires a directory to download data to. Set the environment variable `$PLASMAPAUSE_DATA` prior to importing the module in Python, either in the terminal or inside `/.bashrc`, e.g.:

```
export PLASMAPAUSE_DATA=/path/to/plasmapauses
```

4.10 Usage

On the first run, the database should be downloaded:

```
import ImagePP as ipp

ipp.Download()
```

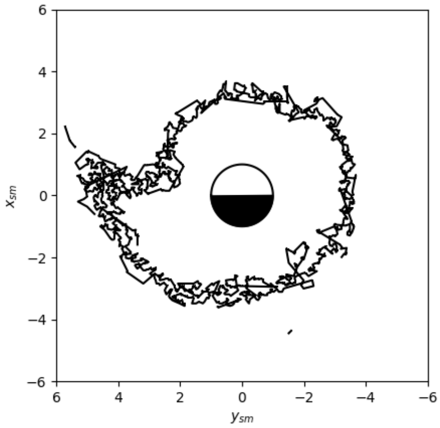
To get a specific plasmapause:

```
#set date in format yyyymmdd
Date = 20010610
ut = 7.0

#get plasmapause coordinates
data = ipp.GetPP(Date,ut)
```

where `data` is a `numpy.recarray` object containing plasmapause coordinates at the equatorial plane in L (`data.L`) and MLT (`data.MLT`) and Cartesian x (`data.x`) and y (`data.y`).

We can also plot that plasmapause (beware, the points are not necessarily stored in order, so results may be wild!):

```
ax = ipp.PlotPP(Date,ut)
```



4.11 PyMess: Download and read MESSENGER data

A Python module for reading the MESSENGER data (or at least some of it)

So far there are some tools for loading and plotting MAG, FIPS and NS data. The spacecraft position can also be retrieved/plotted. A list of bow shock (BS) and magnetopause (MP) crossings based on the work of Winslow et al., 2013 is included, which are used to provide a list of times where MESSENGER is in the solar wind (SW) and the magnetosheath (MSH).

In future I hope to include EPS and XRS submodules too.

4.11.1 Installation

Currently there is no release package on GitHub, nor is there a package to PyPI, so the easiest way to install this module is to download this repository, or clone it using: `git clone https://github.com/mattkjames7/PyMess.git` then to either copy the `PyMess/PyMess` subfolder to your `$PYTHONPATH`, or to create your own Python wheel:

```
cd PyMess
python3 setup.py sdist bdist_wheel
pip3 install dist/PyMess-0.0.1-py3-none-any.whl --user
```

which may, or may not work in the code's current state!

After installation, it would be wise to set up the `$MESSENGER_PATH` environment variable, as this tells the code where to find MESSENGER data.

4.11.2 Submodules

- `FIPS`
- `MAG`
- `NS`
- `Pos`
- `BowShock`
- `Magnetopause`
- `Magnetosheath`
- `ModelField`
- `Tools`

`FIPS` - Fast Imaging Plasma Spectrometer

This module contains routines to read FIPS plasma data. Within the `FIPS` submodule, there is code to convert the PDS (Planetary Data System) data to a more convenient format. The PDS data is stored either in ASCII or binary files. The ASCII files tend to be much larger than necessary, and thus take a long time to read, the binary files are organised in records, which also take a long time to load. The following routines convert both types to a file format where each variable is stored contiguously within the file, allowing for fast reading times and smaller files.

To convert to binary files, run:

```
PyMess.FIPS.PDS.ConvertToBinary()
```

which will scan the `$MESSENGER_PATH/FIPS/PDS` folder for the PDS files.

Another recommended routine combines the new binary files into 1-minute resolution binary files:

```
PyMess.FIPS.Combine60sData()
```

In future, there will be a routine to combine the high time resolution data also.

To load converted data, use the `PyMess.FIPS.ReadFIPS` function, e.g.:

```
data = PyMess.FIPS.ReadFIPS(Date,Type=Type)
```

where `Date` is a 32-bit (or more) integer date in the format yyyymmdd, and `Type` is a string to say the type of data to load, this string can be one of the following:

- 'edr' - To load the EDR data
- 'cdr' - To load the CDR data
- 'ntp' - To load the DDR NTP data
- 'espec' - To load the DDR ESPEC data
- '60' - To load the combined 60s data (default)
- '10' - To load the combined 10s data

`MAG` - Magnetometer

This module contains some basic routines to convert, read and plot the magnetometer data.

To convert PDS data, download PDS data and extract data to `$MESSENGER_PATH/MAG/PDS`. Convert the PDS .TAB files using

```
PyMess.MAG.PDS.ConvertToBinary()
```

which should reduce the size of the dataset from GB to GB.

By default, the magnetometer data used is in MSO coordinates, but we can also rotate the data into a coordinate system more useful for studying ULF waves. In this coordinate system, there is one component parallel to the ambient magnetic field; one oriented in the toroidal/azimuthal direction (eastward); the third component completes the right-handed set and points in the approximately poloidal/radial direction. To convert to these coordinates, run

```
PyMess.MAG.SaveAllRotatedData()
```

To read converted data, for MSO data:

```
data = PyMess.MAG.ReadMagData(Date)
```

For rotated data:

```
data = PyMess.MAG.ReadRotatedData(Date)
```

Also, for magnetopause normal data (based on the magnetopause used by the KT17 magnetic field model):

```
data = PyMess.MAG.MagDataMPN(Date)
```

To plot data:

```
PyMess.MAG.PlotMagData(Date,ut=ut,MagType=MagType)
```

Where, `Date` is a one or two element integer, with the format yyyymmdd, `ut` is a two element list, array or tuple, denoting the time range (from 0 - 24) and `MagType` is a string equal to one of the following: `'MSM'—'Rotated'—'MPN'`.

All of the aforementioned routines have a range of keywords which can be found in their docstrings.

NS - Neutron Spectrometer

This submodule contains some basic routines to convert the NS PDS data to a better binary format, and also to read the new data.

To convert PDS data, place PDS files in `$MESSENGER_PATH/NS/PDS`, where the code will look for them, then run:

```
PyMess.NS.PDS.ConvertToBinary()
```

To read converted data:

4.12 FIPSProtonData: Download and read ANN verified FIPS moments

This Python package was written to provide a simple mechanism for reading and plotting the plasma moments calculated for the MESSENGER FIPS proton spectra.

The moments were found by numerically fitting the kappa distribution function to each proton spectrum using the downhill-simplex method. The quality of each fit was then assessed using neural networks, which classified each spectrum as either "good" or "bad".

WARNING: This is not yet published - do not use for anything serious yet!

4.12.1 Requirements

The following requirements should be installed automatically when using `pip3` to install the package:

- Python 3
- numpy
- matplotlib
- DateTimeTools
- scipy

4.12.2 Installation

For Linux and possibly Mac:

1. Download the latest released wheel here.
2. Open terminal in folder where the wheel was downloaded to.
3. Use `pip3` to install the wheel, e.g.

```
pip3 install FIPSProtonData-0.0.1-py3-none-any.whl --user
```

   where the `--user` flag will install the package locally. Replace the file name above with the name of the downloaded wheel.

Windows:

1. In cmd type `format C:`
2. Install Linux.

4.12.3 Usage

First run

Ideally you would set up an environment variable called `MESSENGER_PATH` which points to a writable directory where you store MESSENGER data, e.g.

```
export MESSENGER_PATH=/data/path/to/MESSENGER
```

This can be done in your .bashrc file, or just run it before starting `python3` or `ipython3`.

To start using the module, open `python3` or `ipython3` and run:

```
import FIPSProtonData as fpd
```

As the module is imported, it will try to find the MESSENGER_PATH if it is set, if not it will use the current working directory.

Loading data

To read in the data simply type in something to the effect of the following:

```
data = fpd.GetData()
```

The first time the GetData function is called, it will search the current MESSENGER_PATH for the data file, then download it when it finds that the file doesn't exist. The data will be downloaded to a subdirectory, $MESSENGER_PATH/FIPS/.

The data object is a numpy.recarray object, containing the following fields:

Fields	dtype	Description
Date	int32	Date in format yyyymmdd
ut	float32	UT in format hh.hhhhh...
mlatn	float32	Magnetic latitude of MESSENGER traced to the north
mlats	float32	Magnetic latitude of MESSENGER traced to the south
latn	float32	Latitude of MESSENGER traced to the north
lats	float32	Latitude of MESSENGER traced to the south
mltn	float32	Magnetic local time of MESSENGER traced to the north
mlts	float32	Magnetic local time of MESSENGER traced to the south
lctn	float32	Local time of MESSENGER traced to the north
lcts	float32	Local time of MESSENGER traced to the south
mlte	float32	MLT of equatorial trace footprint
lshell	float32	L-shell of equatorial footprint
fl_len	float32	Field line length in Rm
x	float32	X-msm coordinate or MESSENGER in Rm
y	float32	Y-msm coordinate or MESSENGER in Rm
z	float32	Z-msm coordinate or MESSENGER in Rm
Loc	U2	String of 'MS','MP','SH','BS','SW','UK'
n	float32	Density in cm^{-3}
t	float32	Temperature in MK
K	float32	Kappa parameter
Bx	float32	X component of local magnetic field
By	float32	Y component of local magnetic field
Bz	float32	Z component of local magnetic field
Rsm	float32	Radial distance of subsolar magnetopause in Rm
Rau	float32	Radial distance of Mercury from the Sun in AU
Class	int8	Indicator of good (==1) or bad (==0) spectrum

The fields in data are easy to access, e.g.:

```
x = data[i].lshell # single element access
x = data.lshell[i] # single element access, equivalent to above
x = data.lshell # array access, x becomes a numpy.ndarray
```

The field line traces which provide the field line length, equatorial footprint, and planetary footprint information were traced using the KT17 magnetic field model (Korth et al., 2015; Korth et al., 2017), the code for which can be found in https://github.com/mattkjames7/KT17.

The Loc variable corresponds to the location of MESSENGER in Mercury's plasma environment at the time of the FIPS measurement. These locations were found using the method described in Winslow et al., 2013.

Plotting data

There are two plotting routines which come with this module: fpd.PlotParameter and fpd.QuickPlot.

QuickPlot will plot *n*, *T*, κ, B_x, B_y, B_z, and $\pm|\mathbf{B}|$ on a single page using:

```
fpd.QuickPlot(Date,ut,ShowClass=True)
```

where Date is either a single integer or a two-element list, tuple, or array of integers in the format *yyyymmdd*, e.g., 12th June 2014 is formatted 20120612. ut is a two-element list, tuple, or array of floating-point values, where the time is formatted in hours, so a time of 13:45:00 would be written as 13.75 (i.e. hh + mm/60 +

ss/3600). `ShowClass` is a Boolean value and shows whether each spectral fit was considered to be good or bad by shading the background either green or red, respectively.

`PlotParameter` is used to plot a single parameter, where one may choose from 'n', 'T', 'K', 'Bx', 'By', 'Bz', 'B', or 'Class', e.g.:

```
fpd.PlotParameter('n',Date,ut)
```

`Date` and `ut` are the same format as for `QuickPlot`, all other keyword arguments can be found in the docstring by typing:

```
fpd.PlotParameter?
```

4.13 `VenusExpress`: Download and read VEX data

Chapter 5

Ground Data and Geomagnetic Indices

5.1 groundmag: Tools for processing and reading ground magnetometer data

5.2 SuperDARN: Simple SuperDARN fitacf reading code

https://github.com/mattkjames7/SuperDARN

The SuperDARN module is for reading and plotting SuperDARN fitacf files. It is a fairly simple tool, but use with caution because there may be some errors...

5.2.1 Installation

This package is not in the PyPI, so manual installation is necessary:

```
#clone the repo
git clone https://github.com/mattkjames7/SuperDARN
cd SuperDARN

#build a Python package
python3 setup.py bdist_wheel

#install it (replace 0.1.0 with whatever version is built)
pip3 install dist/SuperDARN-0.1.0-py3-none-any.whl --user
```

Once installed, the directory used to create the Python wheel file can be deleted. It can be uninstalled using `pip3 uninstall SuperDARN`.

Before running for the first time, a couple of environment variables need to be set up to tell the module where to look for fitacf files and to say where it is able to store some files:

```
#path to where FITACF files are stored
#(this one is specific to SPECTRE)
export FITACF_PATH=/data/sol-ionosphere/fitacf

#path to where this module can create some files
#(this should be a path where you have write access)
export SUPERDARN_PATH=/some/other/path/SuperDARN
```

This module will not currently run on Windows (as far as I am aware) because it requires the compilation of some C++ code which is not yet cross-platform.

Usage

In ipython, the first time this module is imported, it should attempt to download some files from the Radar Software Toolkit (RST) which help in calculating the coordinates of the fields of view of each radar. These files are created in the path defined be the `$SUPERDARN_PATH` variable.

Reading Data

There are a few functions within SuperDARN.Data which provide objects containing data:

```
import SuperDARN as sd

#get the data from a single cell (Radar,Date,ut,Beam,Gate)
cdata = sd.Data.GetCellData('han',20020321,[22.0,24.0],9,25)

#or a whole beam of data (Radar,Date,ut,Beam)
bdata = sd.Data.GetBeamData('han',[20020321,20020322],[22.0,24.0],7)

#data for the whole field of view (Radar,Date,ut)
#in this case, the output is a dict where each key is a beam number
#pointing to a recarray for each beam as produced by GetBeamData
rdata = sd.Data.GetRadarData('han',[20020321,20020322],[22.0,23.0])
```

In the above examples bdata and cdata are numpy.recarray objects, rdata is a dict object containing a numpy.recarray for each beam.

The fitacf data are stored in memory once loaded so that they don't need to be re-read every time the data are requested. To check how much memory is in use and to clear it:

```
#check memory usage in MB
sd.Data.MemUsage()

#clear memory
sd.Data.ClearData()
```

Plotting Data

There are a bunch of very simple plotting functions, e.g.:

```
import matplotlib.pyplot as plt

#create a figure
plt.figure(figsize=(8,11))

#plot the power along a beam
ax0 = sd.Plot.RTIBeam('han',[20020321,20020322],[23.0,1.0],9,[20,35],
                                Param='P_l',ShowScatter=True,fig=plt,
                                maps=[2,3,0,0],scale=[1.0,100.0],zlog=True,
                                cmap='gnuplot')

#the velocity
ax1 = sd.Plot.RTIBeam('han',[20020321,20020322],[23.0,1.0],9,[20,35],Param='V',
                                fig=plt,maps=[2,3,1,0])

#velocity along a range of latitudes at a ~constant longitude of 105
ax2 = sd.Plot.RTILat('han',[20020321,20020322],[23.0,1.0],105.0,Param='V',
                                fig=plt,maps=[2,3,0,1])

#velocity along a range of longitudes at a ~constant latitude of ~70
ax3 = sd.Plot.RTILon('han',[20020321,20020322],[23.0,1.0],70.0,Param='V',
                                fig=plt,maps=[2,3,1,1])

#some specific cells
beams = [1,5,7,2,8,4,9]
gates = [20,26,33,22,25,21,29]
ax4 = sd.Plot.RTI('han',[20020321,20020322],[23.0,1.0],beams,gates,
                                Param='V',fig=plt,maps=[2,3,0,2])

#totally different FOV plot
```

```
ax5 = sd.Plot.FOVData('han',20020321,23.5,Param='V',fig=plt,maps=[2,3,1,2])


plt.tight_layout()
```

which should produce figure 5.1.

Fields of View

These may be wrong. Use with great caution.

The fields of view of each radar are stored as instances of the SuperDARN.FOV.FOVObj objects in memory and can be accessed using GetFOV, e.g.:

```
#get the object from memory
Date = 20020321
fov = sd.FOV.GetFOV('pyk',Date)

#use it to retrieve the FOV in mag coordinates
mlon,mlat = fov.GetFOV(Mag=True,Date=Date)

#plot it
ax = fov.PlotPolar(Background=[0.0,0.2,1.0],Continents=[0.0,1.0,0.2],
                                color='magenta',ShowBeams=False,ShowCells=False,
                                linewidth=2.0,Mag=True,Lon=True)

#add some cells
beams = [1,5,7,2,8,4,9]
gates = [20,26,33,22,25,21,29]
fov.PlotPolarCells(beams,gates,color='red',fig=ax,Mag=True,linewidth=2.0,Lon=True)
```

The above code should look like figure 5.2:

5.3 kpindex: Download the latext Kp indices

Very simple package for obtaining the planetary Kp index data (see https://www.gfz-potsdam.de/en/kp-index/ for more information)

5.3.1 Installation

This package depends on the following:

- numpy
- RecarrayTools
- PyFileIO

which are all available on PyPI.

Installation is simple and can be done in one of four ways:

Method 1

This method simply uses the Python pip3 command to download this module and its dependencies:

```
pip3 install kpindex --user
```

Method 2

This method uses the Python wheel on the "releases" page of this repository. Download the wheel, then install using pip3:

```
pip3 install kpindex-0.0.1-py3-none-any.whl --user
```

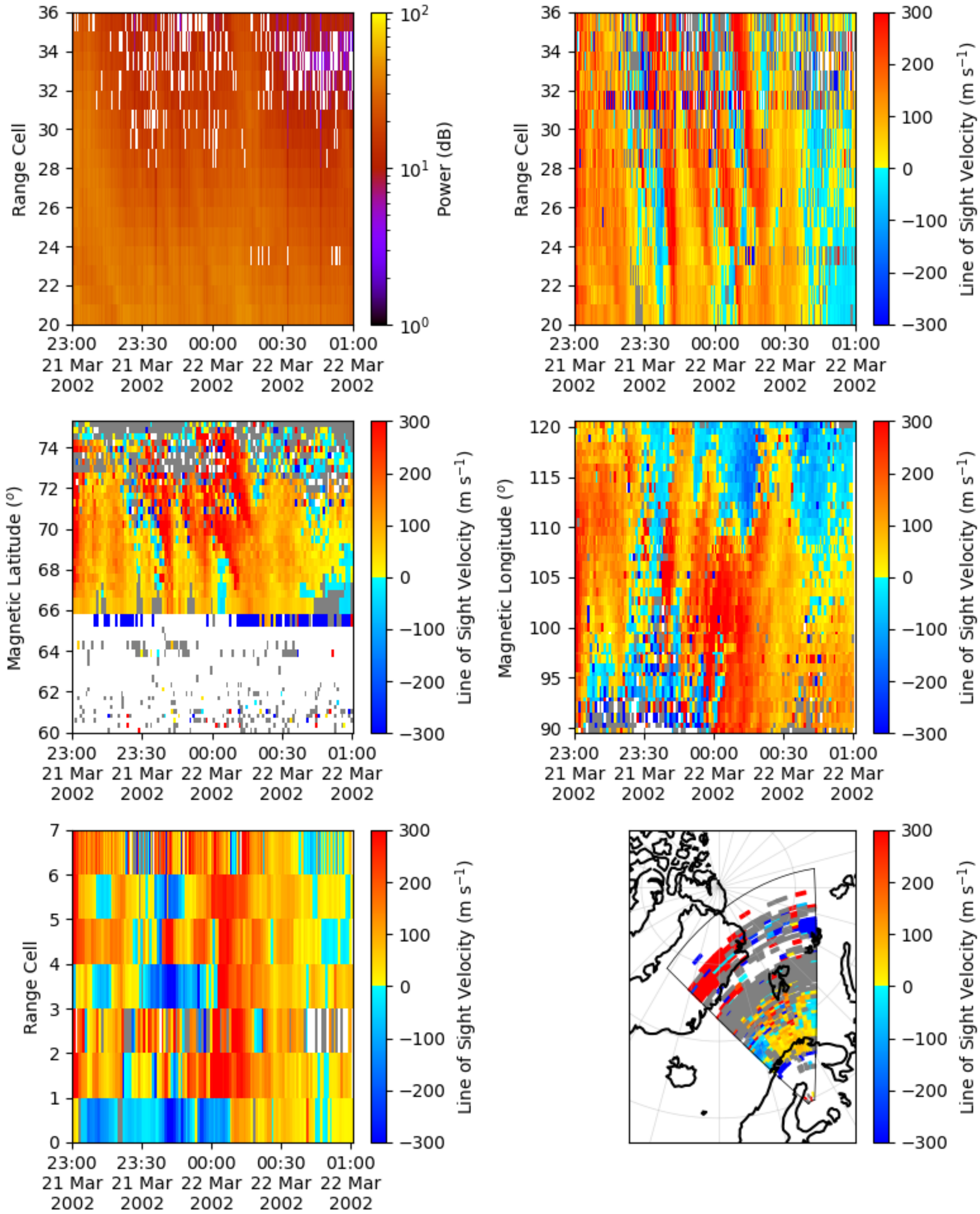


Figure 5.1: Top left: range time intensity (RTI) plot of backscatter power. Top right: RTI plot of line of sight velocity. Mid left: velocity along a line of cells in magnetic longitude. Mid right: velocity along a range of longitudes. Bottom left: velocity of specific range cells. Bottom right: velocity within the field of view plot.

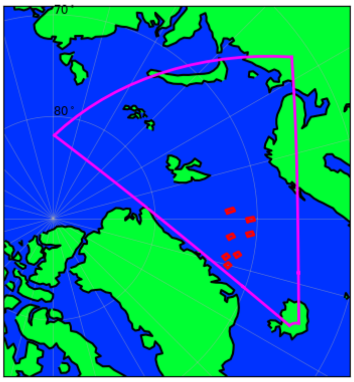


Figure 5.2: SuperDARN field of view plot with specific cells highlighted.

Method 3

Don't trust my prepackaged stuff? OK, clone this repository and build your own:

```
git clone https://github.com/mattkjames7/kpindex.git
cd kpindex
python3 setup.py bdist_wheel
pip3 install dist/kpindex-0.0.1-py3-none-any.whl --user
```

Method 4

So you don't like wheels? Fine. Clone the repository and just move the "kpindex" folder to your `$PYTHONPATH`.

5.3.2 Post-Install

In order for the module to be able to download the Kp index data from the FTP site, you will need to point it in the direction of a directory where you have read and write access using the `$KPDATA`$_PATHenvironmentvariable.Thiscanbedon$

```
export KPDATA_PATH=/path/to/the/data
```

5.3.3 Usage

Using this module is very simple: the first time you run it you will need to update the database (also when you think the database is out of date) e.g.

```
import kpindex
kpindex.UpdateLocalData()
```

It may take a couple of minutes to download the data and convert it, then you are ready to read the data:

```
data = kpindex.GetKp(Date)
```

where Date could be None, in which case ALL of the Kp indices ever will be returned; Date could be a single date in the format yyyymmdd, in which case only Kp indices from that date will be returned; finally, it could be a two-element array/list/tuple containing two dates, in this case, it will return all the indices from the start to the end date.

Enjoy!

5.4 pyomnidata: Download the latext OMNI and solar flux data

Python tool for downloading, converting, and reading OMNI solar wind data.

If you make use of the OMNI data please acknowledge and cite as specified here: https://omniweb.gsfc.nasa.gov/html/citing.html

5.4.1 Installation

Simply install using pip3:

```
pip3 install pyomnidata --user
```

Alternatively install from this repository:

```
git clone https://github.com/mattkjames7/pyomnidata
cd pyomnidata

#EITHER build a wheel and install with pip (better)
python3 setup.py bdist_wheel
pip3 install dist/pyomnidata-1.0.0-py3-none-any.whl --user

#OR directly using setup py (should work, not tested though)
python3 setup.py install --user
```

For this to work properly - you will need to set up the $OMNIDATA$_PATHenvironmentvariabletopointtoafol

```
export OMNIDATA_PATH=/path/to/omni/data
```

5.4.2 Usage

Downloading Data

Download OMNI data like this:

```
import pyomnidata

#download all available data
pyomnidata.UpdateLocalData()
```

Download F10.7 index (solar flux at 10.7 cm):

```
pyomnidata.UpdateSolarFlux(EndDate=2021024)
```

where EndDate is the last date for which you want to request solar flux data. This should be set to a date at least a few days prior to the current date. If you request a date that currently has no available data, the download will fail.

Read Data

Get the OMNI parameters like so:

```
####OMNI parameters####
#Year can either be a single year:
Year = 2001
#or it can be a range:
Year = [2001,2004]

#5 minute resolution data
data = pyomnidata.GetOMNI(Year,Res=5)

#1 minute data
data = pyomnidata.GetOMNI(Year,Res=1)

####solar flux###
#all of the data
data = pyomnidata.GetSolarFlux()

#a single date
data = pyomnidata.GetSolarFlux(Date=20050101)

#a range of dates
data = pyomnidata.GetSolarFlux(Date=[20020101,20020103])
```

The returned data object is a numpy.recarray object which contains all of the OMNI data requested. To see what fields are stored use print(data.dtype.names). The units are as presented here: https://omniweb.gsfc.nasa.gov/html/omni_min_data.html#4b

Plot Data

Use the PlotOMNI function, e.g.:

```
import matplotlib.pyplot as plt

#create a figure
plt.figure(figsize=(11,6))

#plot some stuff in one panel
ax0 = pyomnidata.PlotOMNI(['SymH','SymD'],[20010101,20010120],fig=plt,maps=[2,1,0,0])

#and a second panel
ax1 = pyomnidata.PlotOMNI(['FlowSpeed'],[20010101,20010120],fig=plt,maps=[2,1,1,0])

#fit things a bit nicer
plt.tight_layout()
```

Which should produce something like the following:

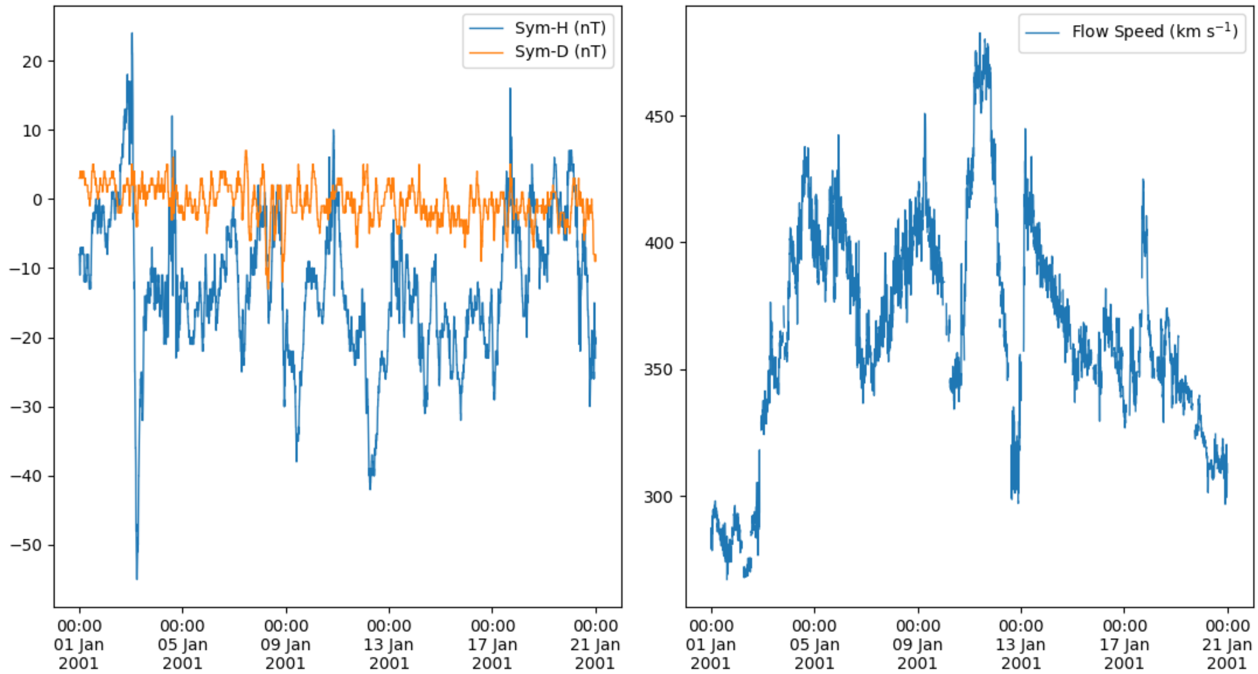


Figure 5.3: Example plot using pyomnidata.